Computer Access Technology Corporation

2403 Walsh Avenue, Santa Clara, CA 95051-1302 Tel: +1/408.727.6600 Fax: +1/408.727.6622

CATC Scripting Language 1.1 Reference Manual for the CATC BPT 1.0

Document Revision 1.0

July 3, 2002

CATC Scripting Language 1.1 Reference Manual for the CATC BPT 1.0, Document Revision 1.0

Product Part Number: 730-0036-00

Document Disclaimer

The information contained in this document has been carefully checked and is believed to be reliable. However, no responsibility can be assumed for inaccuracies that may not have been detected.

CATC reserves the right to revise the information presented in this document without notice or penalty.

Trademarks and Servicemarks

CATC and *BPT* are trademarks of Computer Access Technology Corporation.

Bluetooth is a trademark owned by Bluetooth SIG, Inc. and is used by Computer Access Technology Corporation under license.

All other trademarks are property of their respective companies.

Copyright

TABLE OF CONTENTS

Table of Contents iii

1 Introduction 1
Features of CATC Scripting Language 1
New in CSL Version 1.1 1

2 Values 3
Literals 3
Integers 3
Strings 3
Escape Sequences 4
Lists 4
Raw Bytes 4
Null 4
Variables 5
Global Variables 5
Local Variables 6
Constants 6

3 Expressions 7
`select` expression 7

4 Operators 9
Operations 9
Operator Precedence and Associativity 9

5 Comments 17

6 Keywords 19

7 Statements 21
Expression Statements 21
if Statements 21
if-else Statements 21
while Statements 22
for Statements 22
return Statements 23

Compound Statements 24

8 Preprocessing 27

9 Functions 29

10 Primitives 31
Call() 31
Format() 31
Format Conversion Characters 32
GetNBits() 33
NextNBits() 34
Resolve() 35
Trace() 35

11 BPT Primitives 37
RunTest() 37
Connect() 37
Disconnect() 38
Inquiry() 39
WaitForConnect() 39
L2CAPEchoRequest() 40
MessageBox() 41
Sleep() 41

CHAPTER 1: INTRODUCTION

CATC Scripting Language (CSL) is used to write test scripts for the CATC BPT™, a Bluetooth™ production tester. The BPT uses test scripts to execute Bluetooth commands on devices under test (DUTs). Several test scripts are included with the BPT software installation. They can be used as-is or modified by a test engineer. Additionally, brand new, customized scripts may be written. This allows test engineers to add specialized tests to suit specific production needs.

The test scripts that CATC supplies for the BPT are distributed in the directory in which the BPT application is installed. They are identifiable by the *.script* extension.

If you plan to modify any of the scripts that come with the BPT, it's a good idea to make backups of the original scripts before making changes.

CSL is based on C language syntax, so anyone with a C programming background will have no trouble learning CSL. The simple, yet powerful, structure of CSL also enables less experienced users to easily acquire the basic knowledge needed to start writing custom scripts.

Features of CATC Scripting Language

- Powerful — provides a high-level API while simultaneously allowing implementation of complex algorithms.
- Easy to learn and use — has a simple but effective syntax.
- Self-contained — needs no external tools to run scripts.
- Wide range of value types — provides efficient and easy processing of data.
- General purpose — is integrated into a number of CATC products.

New in CSL Version 1.1

- Compound assignment operators added
- Increment and decrement operators added

CHAPTER 2: VALUES

There are five value types that may be manipulated by a script: **integers**, **strings**, **lists**, **raw bytes**, and **`null`**. CSL is not a strongly typed language. Value types need not be pre-declared. Literals, variables and constants can take on any of the five value types, and the types can be reassigned dynamically.

Literals

Literals are data that remain unchanged when the program is compiled. Literals are a way of expressing hard-coded data in a script.

Integers

Integer literals represent numeric values with no fractions or decimal points. Hexadecimal, octal, decimal, and binary notation are supported:

Hexadecimal numbers must be preceded by `0x`: `0x2A, 0x54, 0xFFFFFF01`

Octal numbers must begin with `0`: `0775, 017, 0400`

Decimal numbers are written as usual: `24, 1256, 2`

Binary numbers are denoted with `0b`: `0b01101100, 0b01, 0b100000`

Strings

String literals are used to represent text. A string consists of zero or more characters and can include numbers, letters, spaces, and punctuation. An *empty string* (`""`) contains no characters and evaluates to false in an expression, whereas a non-empty string evaluates to true. Double quotes surround a string, and some standard backslash (\) escape sequences are supported.

String	Represented text
`"Quote: \"This is a string literal.\""`	`Quote: "This is a string literal."`
`"256"`	`256` **Note that this does not represent the integer 256, but only the characters that make up the number.
`"abcd!$%&*"`	`abcd!$%&*`
`"June 26, 2001"`	`June 26, 2001`
`"[ 1, 2, 3 ]"`	`[ 1, 2, 3 ]`

Table 2.1: Examples of String Literals

Escape Sequences

These are the available escape sequences in CSL:

Character	Escape Sequence	Example	Output
backslash	`\\`	`"This is a backslash: \\"`	This is a backslash: \
double quote	`\"`	`"\"Quotes!\""`	"Quotes!"
horizontal tab	`\t`	`"Before tab\tAfter tab"`	Before tab After tab
newline	`\n`	`"This is how\nto get a newline."`	This is how to get a newline.
single quote	`\'`	`"\'Single quote\'"`	'Single quote'

Table 2.2: Escape Sequences

Lists

A list can hold zero or more pieces of data. A list that contains zero pieces of data is called an *empty list*. An empty list evaluates to false when used in an expression, whereas a non-empty list evaluates to true. List literals are expressed using the square bracket (`[]`) delimiters. List elements can be of any type, including lists.

```
[1, 2, 3, 4]
[]
["one", 2, "three", [4, [5, [6]]]]
```

Raw Bytes

Raw binary values are used primarily for efficient access to packet payloads. A literal notation is supported using single quotes:

```
'00112233445566778899AABBCCDDEEFF'
```

This represents an array of 16 bytes with values starting at `00` and ranging up to `0xFF`. The values can only be hexadecimal digits. Each digit represents a nybble (four bits), and if there are not an even number of nybbles specified, an implicit zero is added to the first byte. For example:

```
'FFF'
```

is interpreted as

```
'0FFF'
```

Null

`Null` indicates an absence of valid data. The keyword `null` represents a literal null value and evaluates to false when used in expressions.

```
result = null;
```

Variables

Variables are used to store information, or data, that can be modified. A variable can be thought of as a container that holds a value.

All variables have names. Variable names must contain only alphanumeric characters and the underscore (_) character, and they cannot begin with a number. Some possible variable names are

```
x
_NewValue
name_2
```

A variable is created when it is assigned a value. Variables can be of any value type, and can change type with re-assignment. Values are assigned using the assignment operator (=). The name of the variable goes on the left side of the operator, and the value goes on the right:

```
x = [ 1, 2, 3 ]
New_value = x
name2 = "Smith"
```

If a variable is referenced before it is assigned a value, it evaluates to null.

There are two types of variables: *global* and *local*.

Global Variables

Global variables are defined outside of the scope of functions. Defining global variables requires the use of the keyword `set`. Global variables are visible throughout a file (and all files that it includes).

```
set Global = 10;
```

If an assignment in a function has a global as a left-hand value, a variable will not be created, but the global variable will be changed. For example

```
set Global = 10;

Function()
{
   Global = "cat";
   Local = 20;
}
```

will create a local variable called `Local`, which will only be visible within the function `Function`. Additionally, it will change the value of `Global` to `"cat",` which will be visible to all functions. This will also change its value type from an integer to a string.

Local Variables

Local variables are not declared. Instead, they are created as needed. Local variables are created either by being in a function's parameter list, or simply by being assigned a value in a function body.

```
Function(Parameter)
{
   Local = 20;
}
```

This function will create a local variable `Parameter` and a local variable `Local`, which has an assigned value of 20.

Constants

A constant is similar to a variable, except that its value cannot be changed. Like variables, constant names must contain only alphanumeric characters and the underscore (_) character, and they cannot begin with a number.

Constants are declared similarly to global variables using the keyword `const`:

```
const CONSTANT = 20;
```

They can be assigned to any value type, but will generate an error if used in the left-hand side of an assignment statement later on. For instance,

```
const constant_2 = 3;

Function()
{
   constant_2 = 5;
}
```

will generate an error.

Declaring a constant with the same name as a global, or a global with the same name as a constant, will also generate an error. Like globals, constants can only be declared in the file scope.

CHAPTER 3: EXPRESSIONS

An expression is a statement that calculates a value. The simplest type of expression is assignment:

```
x = 2
```

The expression `x = 2` calculates 2 as the value of *x*.

All expressions contain operators, which are described in Chapter 4, *Operators*, on page 9. The operators indicate how an expression should be evaluated in order to arrive at its value. For example

```
x + 2
```

says to add 2 to *x* to find the value of the expression. Another example is

```
x > 2
```

which indicates that *x* is greater than 2. This is a Boolean expression, so it will evaluate to either true or false. Therefore, if `x = 3`, then `x > 2` will evaluate to true; if `x = 1`, it will return false.

True is denoted by a non-zero integer (any integer except 0), and false is a zero integer (0). True and false are also supported for lists (an empty list is false, while all others are true), and strings (an empty string is false, while all others are true), and `null` is considered false. However, all Boolean operators will result in integer values.

`select` expression

The `select` expression selects the value to which it evaluates based on Boolean expressions. This is the format for a `select` expression:

```
select {
   <expression1> : <statement1>
   <expression2> : <statement2>
   ...
};
```

The expressions are evaluated in order, and the statement that is associated with the first true expression is executed. That value is what the entire expression evaluates to.

```
x = 10
Value_of_x = select {
   x < 5 : "Less than 5";
   x >= 5 : "Greater than or equal to 5";
};
```

The above expression will evaluate to “Greater than or equal to 5” because the first true expression is `x >= 5`. Note that a semicolon is required at the end of a `select` expression because it is not a compound statement and can be used in an expression context.

There is also a keyword `default`, which in effect always evaluates to true. An example of its use is

```
Astring = select {
   A == 1 : "one";
   A == 2 : "two";
   A == 3: "three";
   A > 3 : "overflow";
   default : null;
};
```

If none of the first four expressions evaluates to true, then `default` will be evaluated, returning a value of `null` for the entire expression.

`select` expressions can also be used to conditionally execute statements, similar to C `switch` statements:

```
select {
   A == 1 : DoSomething();
   A == 2 : DoSomethingElse();
   default: DoNothing();
};
```

In this case the appropriate function is called depending on the value of A, but the evaluated result of the `select` expression is ignored.

CHAPTER 4: OPERATORS

An operator is a symbol that represents an action, such as addition or subtraction, that can be performed on data. Operators are used to manipulate data. The data being manipulated are called *operands*. Literals, function calls, constants, and variables can all serve as operands. For example, in the operation

```
x + 2
```

the variable `x` and the integer `2` are both operands, and `+` is the operator.

Operations

Operations can be performed on any combination of value types, but will result in a null value if the operation is not defined. Defined operations are listed in the Operand Types column of Table 4.2 on page 12. Any binary operation on a null and a non-null value will result in the non-null value. For example, if

```
x = null
```

then

```
3 * x
```

will return a value of 3.

A binary operation is an operation that contains an operand on each side of the operator, as in the preceding examples. An operation with only one operand is called a unary operation, and requires the use of a unary operator. An example of a unary operation is

```
!1
```

which uses the logical negation operator. It returns a value of 0.

The unary operators are `sizeof()`, `head()`, `tail()`, `~` and `!`.

Operator Precedence and Associativity

Operator rules of precedence and associativity determine in what order operands are evaluated in expressions. Expressions with operators of higher precedence are evaluated first. In the expression

```
4 + 9 * 5
```

the `*` operator has the highest precedence, so the multiplication is performed before the addition. Therefore, the expression evaluates to 49.

The associative operator `()` is used to group parts of the expression, forcing those parts to be evaluated first. In this way, the rules of precedence can be overridden. For example,

```
( 4 + 9 ) * 5
```

causes the addition to be performed before the multiplication, resulting in a value of 65.

When operators of equal precedence occur in an expression, the operands are evaluated according to the associativity of the operators. This means that if an operator's associativity is left to right, then the operations will be done starting from the left side of the expression. So, the expression

```
4 + 9 - 6 + 5
```

would evaluate to 12. However, if the associative operator is used to group a part or parts of the expression, those parts are evaluated first. Therefore,

```
( 4 + 9 ) - ( 6 + 5 )
```

has a value of 2.

In Table 4.1, *Operator Precedence and Associativity*, the operators are listed in order of precedence, from highest to lowest. Operators on the same line have equal precedence, and their associativity is shown in the second column.

Operator Symbol	Associativity
`++ --`	Right to left
`[] ()`	Left to right
`~ ! sizeof head tail`	Right to left
`* / %`	Left to right
`+ -`	Left to right
`<< >>`	Left to right
`< > <= >=`	Left to right
`== !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right

Table 4.1: Operator Precedence and Associativity

Operator Symbol									Associativity
=	+=	-=	*=	/=	%=	>>=	<<=	&=	Right to left
				^=	|=				

Table 4.1: Operator Precedence and Associativity (Continued)

Operator Symbol	Description	Operand Types	Result Types	Examples
Index Operator				
[ ]	Index or subscript	Raw Bytes	Integer	`Raw = '001122'` `Raw[1] = 0x11`

		List	Any	`List = [0, 1, 2, 3, [4, 5]]`
				`List[2] = 2`
				`List[4] = [4, 5]`
				`List[4][1] = 5`
				*Note: if an indexed Raw value is assigned to any value that is not a byte (> 255 or not an integer), the variable will be promoted to a list before the assignment is performed.

Associative Operator				
( )	Associative	Any	Any	`( 2 + 4 ) * 3 = 18` `2 + ( 4 * 3 ) = 14`
Arithmetic Operators				
*****	Multiplication	Integer-integer	Integer	`3 * 1 = 3`
/	Division	Integer-integer	Integer	`3 / 1 = 3`
%	Modulus	Integer-integer	Integer	`3 % 1 = 0`
+	Addition	Integer-integer	Integer	`2 + 2 = 4`
		String-string	String	`"one " + "two" = "one two"`
		Raw byte-raw byte	Raw	`'001122' + '334455' = '001122334455'`
		List-list	List	`[1, 2] + [3, 4] = [1, 2, 3, 4]`
		Integer-list	List	`1 + [2, 3] = [1, 2, 3]`
		Integer-string	String	`"number = " + 2 = "number = 2"` *Note: integer-string concatenation uses decimal conversion.
		String-list	List	`"one" + ["two"] = ["one", "two"]`
-	Subtraction	Integer-integer	Integer	`3 – 1 = 2`
Increment and Decrement Operators				
++	Increment	Integer	Integer	*a* = 1 ++*a* = 2 *b* = 1 *b*++ = 1 *Note that the value of *b* after execution is 2.
--	Decrement	Integer	Integer	*a* = 2 --*a* = 1 *b* = 2 *b*-- = 2 *Note that the value of *b* after execution is 1.

Table 4.2: Operators

Operator Symbol	Description	Operand Types	Result Types	Examples
Equality Operators				
==	Equal	Integer-integer	Integer	`2 == 2`
		String-string	Integer	`"three" == "three"`
		Raw byte-raw byte	Integer	`'001122' == '001122'`
		List-list	Integer	`[1, [2, 3]] == [1, [2, 3]]` *Note: equality operations on values of different types will evaluate to false.
!=	Not equal	Integer-integer	Integer	`2 != 3`
		String-string	Integer	`"three" != "four"`
		Raw byte-raw byte	Integer	`'001122' != '334455'`
		List-list	Integer	`[1, [2, 3]] != [1, [2, 4]]` *Note: equality operations on values of different types will evaluate to false.
Relational Operators				
<	Less than	Integer-integer	Integer	`1 < 2`
		String-string	Integer	`"abc" < "def"`
>	Greater than	Integer-integer	Integer	`2 > 1`
		String-string	Integer	`"xyz" > "abc"`
<=	Less than or equal	Integer-integer	Integer	`23 <= 27`
		String-string	Integer	`"cat" <= "dog"`
>=	Greater than or equal	Integer-integer	Integer	`2 >= 1`
		String-string	Integer	`"sun" >= "moon"` *Note: relational operations on string values are evaluated according to character order in the ASCII table.
Logical Operators				
!	Negation	All combinations of types	Integer	`!0 = 1  !"cat" = 0` `!9 = 0  !"" = 1`
&&	Logical AND	All combinations of types	Integer	`1 && 1 = 1  1 && !"" = 1` `1 && 0 = 0  1 && "cat" = 1`
||	Logical OR	All combinations of types	Integer	`1 || 1 = 1  0 || 0 = 0` `1 || 0 = 1  "" || !"cat" = 0`

Table 4.2: Operators (Continued)

Operator Symbol	Description	Operand Types	Result Types	Examples
Bitwise Logical Operators				
~	Bitwise complement	Integer-integer	Integer	`~0b11111110 = 0b00000001`
&	Bitwise AND	Integer-integer	Integer	`0b11111110 & 0b01010101 = 0b01010100`
^	Bitwise exclusive OR	Integer-integer	Integer	`0b11111110 ^ 0b01010101 = 0b10101011`
|	Bitwise inclusive OR	Integer-integer	Integer	`0b11111110 | 0b01010101 = 0b11111111`
Shift Operators				
<<	Left shift	Integer-integer	Integer	`0b11111110 << 3 = 0b11110000`
>>	Right shift	Integer-integer	Integer	`0b11111110 >> 1 = 0b01111111`
Assignment Operators				
=	Assignment	Any	Any	`A = 1` `B = C = A`
+=	Addition assignment	Integer-integer	Integer	*x* `= 1` *x* `+= 1 = 2`
		String-string	String	*a* `= "one "` *a* `+= "two" = "one two"`
		Raw byte-raw byte	Raw	*z* `= '001122'` *z* `+= '334455' = '001122334455'`
		List-list	List	*x* `= [1, 2]` *x* `+= [3, 4] = [1, 2, 3, 4]`
		Integer-list	List	*y* `= 1` *y* `+= [2, 3] = [1, 2, 3]`
		Integer-string	String	*a* `= "number = "` *a* `+= 2 = "number = 2"` *Note: integer-string concatenation uses decimal conversion.
		String-list	List	*s* `= "one"` *s* `+ ["two"] = ["one", "two"]`
-=	Subtraction assignment	Integer-integer	Integer	*y* `= 3` *y* `–= 1 = 2`
***=**	Multiplication assignment	Integer-integer	Integer	*x* `= 3` *x* `*= 1 = 3`
/=	Division assignment	Integer-integer	Integer	*s* `= 3` *s* `/= 1 = 3`
%=	Modulus assignment	Integer-integer	Integer	*y* `= 3` *y* `%= 1 = 0`
>>=	Right shift assignment	Integer-integer	Integer	*b* `= 0b11111110` *b* `>>= 1 = 0b01111111`
<<=	Left shift assignment	Integer-integer	Integer	*a* `= 0b11111110` *a* `<<= 3 = 0b11111110000`

Table 4.2: Operators (Continued)

Operator Symbol	Description	Operand Types	Result Types	Examples
Assignment Operators (continued)				
&=	Bitwise AND assignment	Integer-integer	Integer	*a* = 0b11111110 *a* &= 0b01010101 = 0b01010100
^=	Bitwise exclusive OR assignment	Integer-integer	Integer	*e* = 0b11111110 *e* ^= 0b01010101 = 0b10101011
|=	Bitwise inclusive OR assignment	Integer-integer	Integer	*i* = 0b11111110 *i* |= 0b01010101 = 0b11111111
List Operators				
sizeof()	Number of elements	Any	Integer	sizeof([1, 2, 3]) = 3 sizeof('0011223344') = 5 sizeof("string") = 6 sizeof(12) = 1 sizeof([1, [2, 3]]) = 2 *Note: the last example demonstrates that the sizeof() operator returns the shallow count of a complex list.
head()	Head	List	Any	head([1, 2, 3]) = 1 *Note: the Head of a list is the first item in the list.
tail()	Tail	List	List	tail([1, 2, 3]) = [2, 3] *Note: the Tail of a list includes everything except the Head.

Table 4.2: Operators (Continued)

CHAPTER 5: COMMENTS

Comments may be inserted into scripts as a way of documenting what the script does and how it does it. Comments are useful as a way to help others understand how a particular script works. Additionally, comments can be used as an aid in structuring the program.

Comments in CSL begin with a hash mark (#) and finish at the end of the line. The end of the line is indicated by pressing the Return or Enter key. Anything contained inside the comment delimiters is ignored by the compiler. Thus,

```
# x = 2;
```

is not considered part of the program. CSL supports only end-of-line comments, which means that comments can be used only at the end of a line or on their own line. It's not possible to place a comment in the middle of a line.

Writing a multi-line comment requires surrounding each line with the comment delimiters

```
# otherwise the compiler would try to interpret
# anything outside of the delimiters
# as part of the code.
```

The most common use of comments is to explain the purpose of the code immediately following the comment. For example:

```
# Add a profile if we got a server channel
if(rfChannel != "Failure")
{
   result = SDPAddProfileServiceRecord(rfChannel,
"ObjectPush");
   Trace("SDPAddProfileServiceRecord returned ",
result, "\n");
}
```

CHAPTER 6: KEYWORDS

Keywords are reserved words that have special meanings within the language. They cannot be used as names for variables, constants or functions.

In addition to the operators, the following are keywords in CSL:

Keyword	Usage
`select`	`select` expression
`set`	define a global variable
`const`	define a constant
`return`	`return` statement
`while`	`while` statement
`for`	`for` statement
`if`	`if` statement
`else`	`if-else` statement
`default`	`select` expression
`null`	null value
`in`	input context
`out`	output context

Table 6.1: Keywords

CHAPTER 7: STATEMENTS

Statements are the building blocks of a program. A program is made up of list of statements.

Seven kinds of statements are used in CSL: expression statements, if statements, if-else statements, while statements, for statements, return statements, and compound statements.

Expression Statements

An expression statement describes a value, variable, or function.

```
<expression>
```

Here are some examples of the different kinds of expression statements:

```
Value: x + 3;
Variable: x = 3;
Function: Trace ( x + 3 );
```

The variable expression statement is also called an *assignment statement*, because it assigns a value to a variable.

`if` Statements

An `if` statement follows the form

```
if <expression> <statement>
```

For example,

```
if (3 && 3) Trace("True!");
```

will cause the program to evaluate whether the expression `3 && 3` is nonzero, or True. It is, so the expression evaluates to True and the `Trace` statement will be executed. On the other hand, the expression `3 && 0` is not nonzero, so it would evaluate to False, and the statement wouldn't be executed.

`if-else` Statements

The form for an `if-else` statement is

```
if <expression> <statement1>
else <statement2>
```

The following code

```
if ( 3 - 3 || 2 - 2 ) Trace ( "Yes" );
else Trace ( "No" );
```

will cause “No” to be printed, because `3 - 3 || 2 - 2` will evaluate to False (neither `3 - 3` nor `2 - 2` is nonzero).

`while` Statements

A `while` statement is written as

```
while <expression> <statement>
```

An example of this is

```
x = 2;
while ( x < 5 )
{
   Trace ( x, ", " );
   x = x + 1;
}
```

The result of this would be

```
2, 3, 4,
```

`for` Statements

A `for` statement takes the form

```
for (<expression1>; <expression2>; <expression3>)
<statement>
```

The first expression initializes, or sets, the starting value for *x*. It is executed one time, before the loop begins. The second expression is a conditional expression. It determines whether the loop will continue -- if it evaluates true, the function keeps executing and proceeds to the statement; if it evaluates false, the loop ends. The third expression is executed after every iteration of the statement.

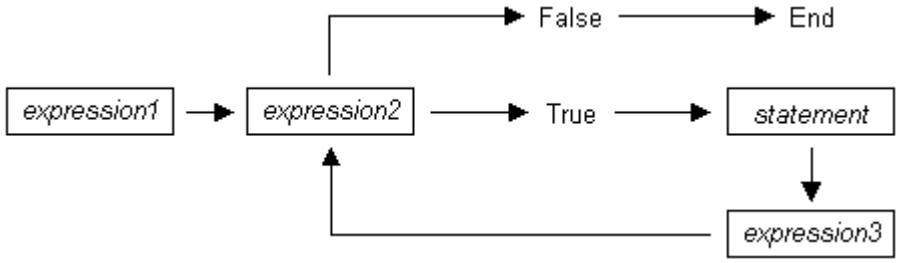

Figure 7-1: Execution of a `for` statement

The example

```
for ( x = 2; x < 5; x = x + 1 ) Trace ( x, "\n" );
```

would output

```
2
3
4
```

The example above works out like this: the expression `x = 2` is executed. The value of *x* is passed to `x < 5,` resulting in `2 < 5`. This evaluates to true, so the statement `Trace (x, "\n" )` is performed, causing 2 and a new line to print. Next, the third expression is executed, and the value of *x* is increased to 3. Now, `x < 5` is executed again, and is again true, so the `Trace` statement is executed, causing 3 and a new line to print. The third expression increases the value of *x* to 4; `4 < 5` is true, so 4 and a new line are printed by the `Trace` statement. Next, the value of *x* increases to 5. `5 < 5` is *not* true, so the loop ends.

`return` Statements

Every function returns a value, which is usually designated in a `return` statement. A `return` statement returns the value of an expression to the calling environment. It uses the following form:

```
return <expression>;
```

An example of a `return` statement and its calling environment is

```
Trace ( HiThere() );
...
HiThere()
{
   return "Hi there";
}
```

The call to the primitive function `Trace` causes the function `HiThere()` to be executed. `HiThere()` returns the string “Hi there” as its value. This value is passed to the calling environment (`Trace`), resulting in this output:

```
Hi there
```

A `return` statement also causes a function to stop executing. Any statements that come after the `return` statement are ignored, because `return` transfers control of the program back to the calling environment. As a result,

```
Trace ( HiThere() );
...
HiThere()
{
   a = "Hi there";
   return a;
   b = "Goodbye";
   return b;
}
```

will output only

```
Hi there
```

because when `return a;` is encountered, execution of the function terminates, and the second return statement (`return b;`) is never processed. However,

```
Trace ( HiThere() );
...
HiThere()
{
   a = "Hi there";
   b = "Goodbye";
   if ( 3 != 3 ) return a;
   else return b;
}
```

will output

```
Goodbye
```

because the `if` statement evaluates to false. This causes the first `return` statement to be skipped. The function continues executing with the `else` statement, thereby returning the value of `b` to be used as an argument to `Trace`.

Compound Statements

A compound statement, or *statement block*, is a group of one or more statements that is treated as a single statement. A compound statement is always enclosed in curly braces (`{}`). Each statement within the curly braces is followed by a semicolon; however, a semicolon is not used following the closing curly brace.

The syntax for a compound statement is

```
{
   <first_statement>;
   <second_statement>;
```

```
   ...
   <last_statement>;
}
```

An example of a compound statement is

```
{
   x = 2;
   x + 3;
}
```

It's also possible to nest compound statements, like so:

```
{
   x = 2;
   {
      y = 3;
   }
   x + 3;
}
```

Compound statements can be used anywhere that any other kind of statement can be used.

```
if (3 && 3)
{
   result = "True!";
   Trace(result);
}
```

Compound statements are required for function declarations and are commonly used in `if`, `if-else`, `while`, and `for` statements.

CHAPTER 8: PREPROCESSING

The preprocessing command `%include` can be used to insert the contents of a file into a script. It has the effect of copying and pasting the file into the code. Using `%include` allows the user to create modular script files that can then be incorporated into a script. This way, commands can easily be located and reused.

The syntax for `%include` is this:

```
%include "includefile.inc"
```

The quotation marks around the filename are required, and by convention, the included file has a `.inc` extension.

The filenames given in the include directive are always treated as being relative to the current file being parsed. So, if a file is referenced via the preprocessing command in a .dec file, and no path information is provided (`%include "file.inc"`), the application will try to load the file from the current directory. Files that are in a directory one level up from the current file can be referenced using `"..\file.inc"`, and likewise, files one level down can be referenced using the relative pathname (`"directory\file.inc"`). Last but not least, files can also be referred to using a full pathname, such as `"C:\global_scripts\include\file.inc"`.

CHAPTER 9: FUNCTIONS

A function is a named statement or a group of statements that are executed as one unit. All functions have names. Function names must contain only alphanumeric characters and the underscore (_) character, and they cannot begin with a number.

A function can have zero or more *parameters*, which are values that are passed to the function statement(s). Parameters are also known as *arguments*. Value types are not specified for the arguments or return values. Named arguments are local to the function body, and functions can be called recursively.

The syntax for a function declaration is

```
name(<parameter1>, <parameter2>, ...)
{
   <statements>
}
```

The syntax to call a function is

```
name(<parameter1>, <parameter2>, ...)
```

So, for example, a function named `add` can be declared like this:

```
add(x, y)
{
   return x + y;
}
```

and called this way:

```
add(5, 6);
```

This would result in a return value of 11.

Every function returns a value. The return value is usually specified using a `return` statement, but if no `return` statement is specified, the return value will be the value of the last statement executed.

Arguments are not checked for appropriate value types or number of arguments when a function is called. If a function is called with fewer arguments than were defined, the specified arguments are assigned, and the remaining arguments are assigned to null. If a function is called with more arguments than were defined, the extra arguments are ignored. For example, if the function `add` is called with just one argument

```
add(1);
```

the parameter `x` will be assigned to 1, and the parameter `y` will be assigned to null, resulting in a return value of 1. But if `add` is called with more than two arguments

```
add(1, 2, 3);
```

`x` will be assigned to 1, `y` to 2, and 3 will be ignored, resulting in a return value of 3.

All parameters are passed by value, not by reference, and can be changed in the function body without affecting the values that were passed in. For instance, the function

```
add_1(x, y)
{
   x = 2;
   y = 3;
   return x + y;
}
```

reassigns parameter values within the statements. So,

```
a = 10;
b = 20;
add_1(a, b);
```

will have a return value of 5, but the values of a and b won't be changed.

The scope of a function is the file in which it is defined (as well as included files), with the exception of primitive functions, whose scopes are global.

Calls to undefined functions are legal, but will always evaluate to null and result in a compiler warning.

CHAPTER 10: PRIMITIVES

Primitive functions are called similarly to regular functions, but they are implemented outside of the language. Some primitives support multiple types for certain arguments, but in general, if an argument of the wrong type is supplied, the function will return null.

Call()

`Call( <function_name` *`string`*`>, <arg_list` *`list`*`> )`

Parameter	Meaning	Default Value	Comments
function_name *string*			
arg_list *list*			Used as the list of parameters in the function call.

Return value

Same as that of the function that is called.

Comments

Calls a function whose name matches the `function_name` parameter. All scope rules apply normally. Spaces in the `function_name` parameter are interpreted as the '_' (underscore) character since function names cannot contain spaces.

Example

```
Call("Format", ["the number is %d", 10]);
```

is equivalent to:

```
Format("the number is %d", 10);
```

Format()

`Format (<format` *`string`*`>, <value` *`string`* `or` *`integer`*`>)`

Parameter	Meaning	Default Value	Comments
format *string*			
value *string* or *integer*			

Return value

None.

Comments

`Format` is used to control the way that arguments will print out. The format string may contain conversion specifications that affect the way in which the arguments in the value string are returned. Format conversion characters, flag characters, and field width modifiers are used to define the conversion specifications.

Example

```
Format("0x%02X", 20);
```

would yield the string `0x14`.

`Format` can only handle one value at a time, so

```
Format("%d %d", 20, 30);
```

would not work properly. Furthermore, types that do not match what is specified in the format string will yield unpredictable results.

Format Conversion Characters

These are the format conversion characters used in CSL:

Code	Type	Output
c	Integer	Character
d	Integer	Signed decimal integer.
i	Integer	Signed decimal integer
o	Integer	Unsigned octal integer
u	Integer	Unsigned decimal integer
x	Integer	Unsigned hexadecimal integer, using "abcdef."
X	Integer	Unsigned hexadecimal integer, using "ABCDEF."
s	String	String

Table 10.1: Format Conversion Characters

A conversion specification begins with a percent sign (%) and ends with a conversion character. The following optional items can be included, in order, between the % and the conversion character to further control argument formatting:

- Flag characters are used to further specify the formatting. There are five flag characters:
  - A minus sign (`-`) will cause an argument to be left-aligned in its field. Without the minus sign, the default position of the argument is right-aligned.
  - A plus sign will insert a plus sign (`+`) before a positive signed integer. This only works with the conversion characters `d` and `i`.

- A space will insert a space before a positive signed integer. This only works with the conversion characters `d` and `i`. If both a space and a plus sign are used, the space flag will be ignored.
- A hash mark (`#`) will prepend a `0` to an octal number when used with the conversion character `o`. If `#` is used with `x` or `X`, it will prepend `0x` or `0X` to a hexadecimal number.
- A zero (`0`) will pad the field with zeros instead of with spaces.

- Field width specification is a positive integer that defines the field width, in spaces, of the converted argument. If the number of characters in the argument is smaller than the field width, then the field is padded with spaces. If the argument has more characters than the field width has spaces, then the field will expand to accommodate the argument.

GetNBits()

`GetNBits (<bit_source` *`list`* `or` *`raw`*`>, <bit_offset` *`integer`*`>, <bit_count` *`integer`*`>)`

Parameter	Meaning	Default Value	Comments
bit_source *list*, *raw*, or *integer*			Can be an integer value (4 bytes) or a list of integers that are interpreted as bytes.
bit_offset *integer*	Index of bit to start reading from		
bit_count *integer*	Number of bits to read		

Return value

None.

Comments

Reads `bit_count` bits from `bit_source` starting at `bit_offset`. Will return null if `bit_offset` + `bit_count` exceeds the number of bits in `bit_source`. If `bit_count` is 32 or less, the result will be returned as an integer. Otherwise, the result will be returned in a list format that is the same as the input format. `GetNBits` also sets up the bit data source and global bit offset used by `NextNBits` and `PeekNBits`. Note that bits are indexed starting at bit 0.

Example

```
raw = 'F0F0';     # 1111000011110000 binary
result = GetNBits ( raw, 2, 4 );
Trace ( "result = ", result );
```

The output would be

```
result = C        # The result is given in
hexadecimal. The result in binary is 1100.
```

In the call to `GetNBits`: starting at bit 2, reads 4 bits (1100), and returns the value 0xC.

NextNBits()

`NextNBits (<bit_count` *integer*`>)`

Parameter	Meaning	Default Value	Comments
bit_count *integer*			

Return value

None.

Comments

Reads `bit_count` bits from the data source specified in the last call to `GetNBits`, starting after the last bit that the previous call to `GetNBits` or `NextNBits` returned. If called without a previous call to `GetNBits`, the result is undefined. Note that bits are indexed starting at bit 0.

Example

```
raw = 'F0F0';# 1111000011110000 binary
result1 = GetNBits ( raw, 2, 4 );
result2 = NextNBits(5);
result3 = NextNBits(2);
Trace ( "result1 = ", result1, " result2 = ", result2,
" result3 = ", result3 );
```

This will generate this trace output:

```
result1 = C result2 = 7 result3 = 2
```

In the call to `GetNBits`: starting at bit 2, reads 4 bits (1100), and returns the value 0xC.

In the first call to `NextNBits`: starting at bit 6, reads 5 bits (00111), and returns the value 0x7.

In the second call to `NextNBits`: starting at bit 11 (= 6 + 5), reads 2 bits (10), and returns the value 0x2.

Resolve()

`Resolve( <symbol_name` *`string`*`> )`

Parameter	Meaning	Default Value	Comments
symbol_name *string*			

Return value

The value of the symbol. Returns null if the symbol is not found.

Comments

Attempts to resolve the value of a symbol. Can resolve global, constant and local symbols. Spaces in the `symbol_name` parameter are interpreted as the '_' (underscore) character since symbol names cannot contain spaces.

Example

```
a = Resolve( "symbol" );
```

is equivalent to:

```
a = symbol;
```

Trace()

`Trace( <arg1` *`any`*`>, <arg2` *`any`*`>, ...)`

Parameter	Meaning	Default Value	Comments
arg *any*			The number of arguments is variable.

Return value

None.

Comments

The values given to this function are given to the debug console.

Example

```
list = ["cat", "dog", "cow"];
Trace("List = ", list, "\n");
```

would result in the output

```
List = [cat, dog, cow]
```

CHAPTER 11: BPT PRIMITIVES

RunTest()

```
RunTest( Address )
```

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to run test on		

Return value

- "Success"
- Error message

Comments

This is the entry point into a script. When a script is run, the script's `RunTest()` function will be called. Include this command at the beginning of every script.

Example

```
RunTest( Address )
{
   # include body of script here
}
```

Connect()

```
Connect( Address )
```

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to connect with		

Return value

- "Success"
- "Failure"
- "Disconnection in progress"
- "Already connected"

Comments

Establishes an ACL connection with the specified device.

Example

```
result = Connect( Address );

if( result != "Success" )
{
   MessageBox( "Failed to connect!" );
}
```

Disconnect()

```
Disconnect( Address )
```

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to disconnect from		

Return value

- "Success"
- "Failure"
- "Disconnection in progress"

Comments

Closes the ACL connection with the specified device.

Example

```
result = Disconnect( Address );

if( result != "Success" )
{
   MessageBox( "Failed to disconnect!" );
}
```

Inquiry()

```
Inquiry( IAC, Timeout )
```

Parameter	Meaning	Default Value	Comments
IAC	Inquiry Access Code	GIAC	"GIAC", or a 32-bit integer value
Timeout	Timeout in units of 1.2 seconds		

Return value

- Array of Bluetooth addresses that were found during the inquiry.
- "Failure"
- "Inquiry in progress"

Comments

Calling `Inquiry()` will block for the duration specified by `Timeout`. The function returns an array of devices that were found during the inquiry. These can be addressed individually. The current version of BPT hardware only supports GIAC inquiries.

Example

```
# Use default parameters
Devices = Inquiry( );

Trace( "First device was: ", Devices[ 0 ] );
```

WaitForConnect()

```
WaitForConnect( Address, TimeoutMS )
```

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to wait for connection with		
TimeoutMS	Timeout in milliseconds	10000 (10 seconds)	

Return value

- "Success"

- “Failure”
- “Already connected”

Comments

Waits for an incoming ACL connection from a specified device for a specified time.

Example

```
result = WaitForConnect( Address );
```

L2CAPEchoRequest()

```
L2CAPEchoRequest( Address, EchoData )
```

Parameter	Meaning	Default Value	Comments
Address	Bluetooth address of device to request echo data from		
EchoData	Test data string		String should not exceed 44 characters

Return value

- “Success”, EchoData
- “Failure”
- “Not connected”
- “Invalid parameter”

Comments

Sends an Echo Request to the L2CAP protocol on the specified remote device.

Example

```
Result = L2CAPEchoRequest( Address, "Maximum
bogosity" );

if( Result[ 0 ] == "Success" )
{
   MessageBox( Result[ 1 ], "Echo request result" );
}
```

MessageBox()

```
MessageBox( Message, Caption )
```

Parameter	Meaning	Default Value	Comments
Message	Text to display in the message box		
Caption	Caption of the message box	"Script Message"	

Return value

None.

Comments

Bring up a simple message box function with one "OK" button. This is a good way to pause execution of the script or indicate errors.

Example

```
MessageBox( "Failed to connect",
  "Connection Failure" );
```

Sleep()

```
Sleep( TimeInMS )
```

Parameter	Meaning	Default Value	Comments
TimeInMS	Time in milliseconds		

Return value

None.

Comments

Delays program execution for a specified number of milliseconds.

Example

```
Sleep( 1000);  # Sleep for one second
```