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 Reference Manual for FireInspector™

Document Revision 1.0

December 17, 2001 730-0026-00

CATC Scripting Language Reference Manual for FireInspector, Document Revision 1.0

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 FireInspector are trademarks of Computer Access Technology Corporation.

All other trademarks are property of their respective companies.

Copyright

Copyright 2001, Computer Access Technology Corporation (CATC). All rights reserved.

This document may be printed and reproduced without additional permission, but all copies should contain this copyright notice.

Part number: 730-0026-00

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

CATC Scripting Language (CSL) was developed to create scripts that allow users to do file-based decoding with CATC analyzers. CSL is used to edit CATC Decode Scripting (CDS) files, which are pre-written decoder scripts supplied by CATC. These script-based decoders can be modified by the users or used as-is. Additionally, users can create brand new CDS files. This document describes the basics of CSL syntax and also defines FireInspector-specific contexts and primitives.

Decoder scripts for FireInspector are distributed in the `\Scripts` folder of the FireInspector installation directory. They are identifiable by the .dec extension. These scripts are tools to decode and display transactions. Users can also add entirely new, customized decoders to fit their own, specific development needs. FireInspector looks in the `\Scripts` directory and automatically loads all of the .dec files that it finds. To prevent a particular decoder from being loaded, change its extension to something other than .dec or move it out of the `\Scripts` directory.

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.
- Used to create built-in script-based decoders for analyzers.
- May be used to write custom decoders.
- General purpose -- is integrated in a number of CATC products.

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 11. 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 the following table, 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
`[]  ()`	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
`=`	Right to left

Table 4.1: Operator Precedece and Associativity

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`

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 Operator				
=	Assignment	Any	Any	`A = 1` `B = C = A`
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.

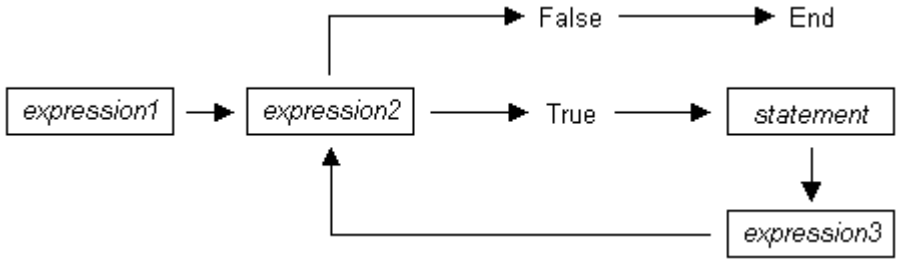

Figure 7-1: Execution of a `for` statement

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: CONTEXT

The context is the mechanism by which transaction data is passed in and out of the scripts. There is an output context that is modified by the script, and there are possibly multiple input contexts that the script will be invoked on separately.

A context serves two roles: firstly, it functions as a symbol table whose values are local to a particular transaction; secondly, it functions as an interface to the application. Two keywords are used to reference symbols in the context: `in` and `out`. Dot notation is used to specify a symbol within a context:

```
out.symbol = "abcd";
out.type = in.type;
```

The output context can be read and written to, but the input context can only be read. Context symbols follow the same rules as local variables: they are created on demand, and uninitialized symbols always evaluate to null.

When a script is first invoked, it is given an input context that corresponds to a packet or transaction that is a candidate for being a part of a larger transaction. The output context is initially empty. It is the script's job to examine the input context and decide if it qualifies for membership in the type of transaction that the script was designed to decode. If it qualifies, the appropriate values will be decoded and put in the output context symbol table, and if the transaction is complete, it will be done. If the transaction is not complete, the script will indicate this to the application based on its return value, and will be invoked again with the same output context, but a new input context. The script then must decide if this new input context is a member of the transaction, and keep doing this until the transaction is complete.

In order to accomplish all this, state information should be placed in the output context. It should be possible to use the output context of one transaction as an input context to another transaction.

CHAPTER 10: TRANSACTION AND PACKET CONTEXT FIELDS

This chapter describes the transaction and packet context fields (symbols) that are defined in FireInspector. These fields define the output context for each built-in transaction or packet type. These output contexts are then used as input to script decoders. A script decoder declares the transaction types that it wishes to examine by using the `InputType` module variable (for more information, see "`InputType`" on page 66).

Transaction Context Fields

The transaction context fields are listed by type of transaction.

1394 Transactions

These transactions require the module data `InputType = "1394 Transaction"`.

`data_len`: *Integer*. Length of data payload in bytes. Taken from the packet header.

`Tcode`: *Integer*. `Tcode` value of the transaction's request packet.

`Requester`: *Integer*. Node ID of the transaction initiator.

`Responder`: *Integer*. Node ID of the transaction target.

`Rcode`: *Integer*. The result of the transaction. This could be the `rcode` field from a response packet or from the `ack` value of the request packet. The meaning of the values are the same as for a packet `rcode`.

`Payload`: *Raw bytes*. The data payload.

`address`: *Raw bytes*. The 48-bit address.

`BusResetOccurred`: *Integer*. A non-zero value means that a 1394 bus reset occurred since the last 1394 transaction; otherwise, the value is zero.

IPv4 over 1394 Transactions

These transactions require the module data `InputType = "IPv4 / 1394"`.

`Data`: *Raw bytes*. The data payload of the transaction.

`DataLength`: *Integer*. The length of the payload in bytes.

`Source_Node`: *Integer*. The 1394 node ID of the source node.

`Target_Node`: *Integer*. The 1394 node ID of the target node.

`LF`: *Integer*. The `LF` field of the encapsulation header.

`datagram_size`: *Integer*. The `datagram_size` field of the encapsulation header.

`offset`: *Integer*. The `offset` field of the encapsulation header.

`DGL`: *Integer*. The `DGL` field of the encapsulation header.

`ether_type`: The `ether_type` of the encapsulation header.

IP Datagram Transactions

These transactions require the module data `InputType = "IP Datagram"`.

`Data`: *Raw bytes*. Fully assembled datagram buffer.

`DataLength`: *Integer*. Length of `Data`, in bytes.

`Payload`: *Raw bytes*. Payload of the datagram (essentially `Data` with the datagram header stripped off.)

`PayloadLength`: *Integer*. Length of `Payload` buffer, in bytes.

Datagram header fields

The rest of the fields are integers taken directly from the Datagram header defined in RFC-791.

- `Version`
- `IHL`
- `Type_of_Service`
- `Total_Length`
- `Identification`
- `Flags`
- `DF`
- `MF`
- `Fragment_Offset`
- `Time_to_Live`
- `Protocol`
- `Header_Checksum`
- `Source_Address`

- `Dest_Address`

IP Protocol Transactions

These transactions require the module data `InputType = "IP Protocol"`.

`Data`: *Raw bytes*. The fully assembled buffer (includes protocol header).

`DataLength`: *Integer*. Length of `Data`, in bytes.

`Payload`: *Raw bytes*. Same as `Data`, but with the protocol header stripped off.

`PayloadLength`: *Integer*. Length of `Payload` buffer, in bytes.

TCP header fields

The following fields are valid for TCP headers. These are all integer values and come directly from RFC-793.

- `Source_Port`
- `Dest_Port`
- `Sequence_Number`
- `Acknowledgment_Number`
- `Data_Offset`
- `URG`
- `ACK`
- `EOL`
- `RST`
- `SYN`
- `FIN`
- `Window`
- `Checksum`
- `Urgent_Pointer`

UDP header fields

The following fields are valid for UDP headers. These are all integer values and come directly from RFC-768.

- `Source_Port`
- `Dest_Port`

- `Length`
- `Checksum`

ICMP header fields

The following fields are valid for ICMP headers. These are all integer values and come directly from RFC-792.

- `Type`
- `Code`
- `Checksum`
- `Pointer`
- `Ident`
- `Seq`
- `Originate_Timestamp`
- `Receive_Timestamp`
- `Transmit_Timestamp`

FCP Transactions:

These transactions require the module data
`InputType = "FCP Transaction"`.

Fields in all FCP transactions:

`CTS`: CTS code of the transaction.

`Controller`: Node ID of the requesting node.

`Target`: Node ID of the responding node.

Fields defined for AV/C transactions:

`ctype`: *Integer.* AV/C `ctype`.

`response`: *Integer.* AV/C `response`.

`su_type`: *Integer.* AV/C `su_type`. Includes extended types.

`su_id`: *Integer.* AV/C `su_id`. Includes extended ids.

`Opcode`: *Integer.* AV/C `opcode`.

Packet Context Fields

A note about using packets as input to a script:

When a file is loaded, FireInspector automatically decodes 1394 transactions but does not display them. A packet (or sub-transaction) is only allowed to be a member of one higher level transaction. This means that all of the packets that belong to 1394 transactions will not be handed to a script that takes packets as input.

Fields defined for packet-level transactions in FireInspector

These transactions require the module data `InputType = "Packet"`.

`PacketType`: *Integer*. A value which identifies the type of packet:

Possible values:

- `0x00 -- unknown`
- `0x01 -- write data quadlet request`
- `0x02 -- write data block request`
- `0x03 -- read data quadlet request`
- `0x04 -- read data block request`
- `0x05 -- lock request`
- `0x06 -- write response`
- `0x07 -- read data quadlet response`
- `0x08 -- read data block response`
- `0x09 -- lock response`
- `0x0A -- cycle start`
- `0x0B -- isochronous data block`
- `0x0C -- bad tcode`
- `0x0D -- asynchronous stream`
- `0x0E -- Global Asynchronous Stream Packet (GASP)`
- `0x10 -- config`
- `0x11 -- extended`
- `0x12 -- link on`
- `0x13 -- self id`
- `0x15 -- bad phy packet`

- `0x16 -- ping`

Note: values >= `0x10` are `phy` packets.

`raw_data`: *Raw bytes*. Raw unparsed packet header data.

`Payload`: *Raw bytes*. Data payload of packet. This field will also contain a single quadlet in the case of quadlet format packets. This field will be null if packet contains no data.

`ack`: *Integer*. Acknowlegement value (without the error-checking bits). This field will be null if one was not received.

Standard 1394 packet field names

The rest of these fields are simply the packet header fields right out of the 1394 specifications. For descriptions of each field, please refer to the 1394 specification. If a packet type does not have a particular field, the field will be set to null.

`tcode`: *Integer*.

`header_CRC`: *Integer*.

`pri`: *Integer*.

`rt`: *Integer*.

`tl`: *Integer*.

`source_ID`: *Integer*.

`destination_ID`: *Integer*.

`destination_offset`: *Raw bytes*.

`rcode`: *Integer*.

`extended_tcode`: *Integer*.

`data_length`: *Integer*.

`quadlet_data`: *Integer*.

`channel`: *Integer*.

`tag`: *Integer*.

`sy`: *Integer*.

`cycle_time_data`: *Integer*.

Example

The following example is taken from the file IPProtocol.dec, which is included with the FireInspector installation.

```
if ( in.Payload == null )
   return Reject();

if ( in.Version != 4 )
   return Reject();

...

if ( out.Identification == null )
{
   out.Identification = in.Identification;
}
else
{
   if ( out.Identification != in.Identification )
   {
      return Reject();
   }
}
```

CHAPTER 11: 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 12: 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 12.1: 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 = 'FOFO';     # 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 = 'FOFO';# 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 13: DECODER PRIMITIVES

Abort()

`Abort()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

An integer that should be passed back to the application unchanged.

Comments

Called when an input context renders the currently pending transaction done, but is not itself a member of that transaction. An example would be an input transaction that represents some sort of reset condition that renders all pending transactions invalid. The input transaction is not consumed by this action and will go on to be considered for other pending transactions.

Example

```
if ( IsReset )
return Abort();
```

AddCell()

`AddCell(<name` *`string`*`>, <value` *`string`*`>, <description` *`string`* `or` *`null`*`>, <color` *`integer`* `or` *`list`*`>, <additional_info` *`any`*`>)`

Parameter	Meaning	Default Value	Comments
name *string*			Displays in the name field of the cell.
value s*tring*			Displays in the value field of the cell.
description *string* or *null*			Displays in tool tip.
color *integer* or *list*		If not specified, a default color is used	Color can be specified as either a packed color value in an integer, or as an array of RGB values ranging from 0-255. Displays in the name field of the cell.

Parameter	Meaning	Default Value	Comments

additional_info *any*			Used to create special cells or to modify cell attributes. The values are predefined constants, and zero or more of them may be used at one time. Possible values are:
			`_COLLAPSED`
			`_ERROR`
			`_EXPANDED`
			`[_FIXEDWIDTH, w]`
			`_HIDDEN`
			`_MONOCOLOR`
			`_MONOFIELD`
			`_SHOWN (default)`
			`_WARNING`

Return value

None.

Comments

Adds a display cell to the current output context. Cells are displayed in the order that they are added. The name and value strings are displayed directly in the cell.

Example

```
# Create a regular cell named Normal with a value
"Cell" and tool tip "Normal cell":

AddCell( "Normal", "Value1", "Normal cell" );

# Use the _MONOCOLOR value in the additional_info
parameter to create a cell with a color value of
0x881122 in both the name and value fields:

AddCell( "MonoColor", "Value2", "MonoColor cell",
0x881122, _MONOCOLOR );

# Use the _MONOFIELD value to create a cell with only
a name field:

AddCell( "MonoField", "Value3", "MonoField cell",
[255, 200, 200], _MONOFIELD );

# Use the _ERROR value to create a cell with a red
value field:

AddCell( "Error", "Value4", "Error cell", 0xcc1155,
_ERROR );

# Use the _WARNING value to create a cell with a yellow
value field:
```

```
AddCell( "Warning", "Value5", "Warning cell",
0x00BB22, _WARNING );

# Use the [_FIXEDWIDTH, w] value to create a cell with
a fixed width of 20 in conjuction with the error value
to create a fixed width cell with a red value field:

AddCell( "Fixed Width 20", "Value6", "Fixed Width and
Error cell", 0x001122, [_FIXEDWIDTH, 20], _ERROR );
```

The output of the example is:

Figure 13-1: Example output for `AddCell`

AddDataCell()

`AddDataCell(<data_value` *`raw`*`, `*`list`*` or `*`integer`*`>, <additional_info` *`any`*`>, ...)`

Parameter	Meaning	Default Value	Comments
data_value *raw*, *list*, or *integer*			Interpreted the same way as `GetNBits` interprets `data_source`

additional_info *any*			Used to create special cells or to modify cell attributes. Possible values are:
			`_BYTES`
			`_COLLAPSED`
			`_DWORDS`
			`_EXPANDED`
			`_HIDDEN`
			`_SHOWN (default)`

Return value

None.

Comments

Creates an expandable/collapsible cell for viewing raw data such as data payloads. Data can be raw bytes, an integer, or a list. If an integer is used, it will be interpreted as 4 bytes of data. Specifying `_BYTES` or `_DWORDS` in an `additional_info` field will force data to be interpreted as bytes or quadlets. `_COLLAPSED`, `_EXPANDED`, `_HIDDEN` and `_SHOWN` are all interpreted the same is in a regular `AddCell` call.

Example

```
# Creates a data cell with 2 dwords (32-bit integers)
of data.

AddDataCell( '0123456789ABCDEF', _DWORDS );

# Creates a data cell with 4 bytes.  Integer data
values are always interpreted as 32 bits of data.

AddDataCell( 0x11223344, _BYTES );
```

The output of the example is:

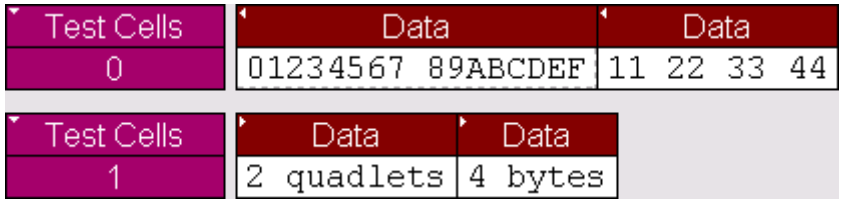

Figure 13-2: Example output for `AddDataCell`

AddEvent()

`AddEvent(<Group` *`string`*`>, <Value` *`string`*`> )`

Parameter	Meaning	Default Value	Comments
Group *string*	The name of the group		Corresponds to the name of a field that might be encountered while decoding.
Value *string*	A value that will be associated with the group		Corresponds to a field value that might be encountered while parsing.

Return value

None.

Comments

Events are used for transaction searching and for transaction summary. This function is only effective when called during the `ProcessData()` phase of decoding. Event groups and values are stored globally for transaction levels and new ones are created as they are encountered. Each transaction contains information as to which events were associated with it.

Example

```
AddEvent( "DataLength", Format( "%d",
out.DataLength ));
```

AddSeparator()

`AddSeparator(<additional_info *any*>, ...)`

Parameter	Meaning	Default Value	Comments

additional_info *any*			Used to create special cells or to modify cell attributes. The values are predefined constants. Possible values are:
			`_COLLAPSED`
			`_EXPANDED`
			`_HIDDEN`
			`_SHOWN (default)`

Return value

None.

Comments

Creates a separator cell. `_COLLAPSED`, `_EXPANDED`, `_HIDDEN`, and `_SHOWN` are all interpreted the same is in a regular `AddCell` call.

Example

```
AddCell( "Stuff", "Things" );

# AddSeparator adds a space between the previous and
subsequent cells.

AddSeparator();

AddCell( "More stuff", "More things" );
```

The output of the example is:

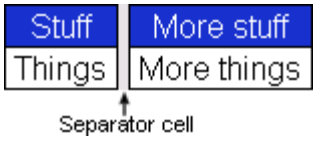

Figure 13-3: Separator cell example

BeginCellBlock()

`BeginCellBlock(<name *string*>, <value *string*>, <description *string* or *null*>, <color *integer* or *list*>, <additional_info *any*>)`

Parameter	Meaning	Default Value	Comments
name *string*			Displays in the name field of the cell.

Parameter	Meaning	Default Value	Comments
value *string*			Displays in the value field of the cell.
description *string* or *null*			Displays in tool tip.
color *integer* or *list*		If not specified, a default color is used	Color can be specified as either a packed color value in an integer, or as an array of RGB values ranging from 0-255. Displays in the name field of the cell.

additional_info *any*			Used to create special cells or to modify cell attributes. The values are predefined constants, and zero or more of them may be used at one time. Possible values are:
			`[_BLOCKNAME, x]`
			`_COLLAPSED`
			`_ERROR`
			`_EXPANDED`
			`[_FIXEDWIDTH, w]`
			`_HIDDEN`
			`_MONOCOLOR`
			`_MONOFIELD`
			`_SHOWN (default)`
			`_WARNING`

Return value

None.

Comments

Begins a cell block and adds a block header cell. This is a special cell that can be collapsed and expanded. The collapsed/expanded state of this cell affects cells in the group according to their `_COLLAPSED`, `_EXPANDED` attributes. All calls to `AddCell` after a call to `BeginCellBlock()` will put the new cells into this group until a call to `EndCellBlock` is made.

Cell blocks can be nested.

Example

```
# Begin the 'red' group.  For clarity these cells will
be red:

BeginCellBlock( "Red Group", null, null, 0x0000ff,
_MONOFIELD );

# This cell will be displayed when the red group is in
the expanded state:

AddCell( "Red is", "Expanded", null, 0x0000ff,
_EXPANDED );
```

```
# This cell will be displayed when the red group is
collapsed:

AddCell( "Red is", "Collapsed", null, 0x0000ff,
_COLLAPSED );

# This begins the nested blue group.  Nothing in the
blue group will be displayed unless the red group is
expanded:

BeginCellBlock( "Blue Group", null, null, 0xff0000,
_MONOFIELD, _EXPANDED, [_BLOCKNAME, "BlockName"] );

# This cell is only displayed when the blue group is
visible and expanded:

AddCell( "Blue is", "Expanded", null, 0xff0000,
_EXPANDED );

# This cell is also only displayed when the blue group
is visible and expanded:

AddCell( "Blue", "Too", null, 0xff0000, _EXPANDED );

# This cell is only displayed when the blue group is
visible and collapsed:

AddCell( "Blue is", "Collapsed", null, 0xff0000,
_COLLAPSED );

# This ends the blue group.

EndCellBlock();

# Cells with the _SHOWN attribute are always
displayed.  This is the default:

AddCell( "Always", "Shown", null, 0x0000ff, _SHOWN );

# This cell will never be displayed.  In a real script
this would be driven by a variable:

AddCell( "Never", "Shown", null, 0x0000ff, _HIDDEN );

# This ends the red group.

EndCellBlock();
```

The output of the example is:

Figure 13-4: Example output for `BeginCellBlock` with red group collapsed

Figure 13-5: Example output for `BeginCellBlock` with red group expanded and blue group collapsed

Figure 13-6: Example output for `BeginCellBlock` with red group expanded and blue group expanded

Complete()

`Complete()`

Parameter	Meaning	Default Value	Comments

Return value

An integer that should be passed back to the application unchanged.

Comments

This should be called when it has been decided that an input context has been accepted into a transaction, and that the transaction is complete. The return value of this function should be passed back to the application from the `ProcessData` function. This function could be used to associate the input context with the output context.

Example

```
if ( done )
return Complete();
```

EndCellBlock()

`EndCellBlock()`

Parameter	Meaning	Default Value	Comments

Return value

None.

Comments

```
Ends a cell block that was started with
BeginCellBlock().
```

Example

See `BeginCellBlock()`.

GetBitOffset()

`GetBitOffset()`

Parameter	Meaning	Default Value	Comments
N/A			

Return value

None.

Comments

Returns the current bit offset that is used in `NextNBits` or `PeekNBits`.

Example

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

The example generates this Trace output:

```
Offset = D
```

PeekNBits()

`PeekNBits(<bit_count integer>)`

Parameter	Meaning	Default Value	Comments
bit_count *integer*			

Return value

None.

Comments

Reads `bit_count` bits from the data source. The difference between `PeekNBits` and `NextNBits` is that `PeekNBits` does not advance the global bit offset. `PeekNBits` can be used to make decisions about how to parse the next fields without affecting subsequent calls to `NextNBits`. If `PeekNBits` is called without a prior call to `GetNBits`, the result is undefined. Note that bits are indexed starting at bit 0.

Example

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

This will generate this Trace output:

```
result1 = C result2 = 7 result3 = 0
```

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

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

In the call to `NextNBits`: starting at bit 6, reads 2 bits (00), and returns the value 0x0.

Pending()

`Pending()`

Parameter	Meaning	Default Value	Comments

Return value

An integer that should be passed back to the application unchanged.

Comments

This should be called when it has been decided that an input context has been accepted into a transaction, but that the transaction still requires further input to be complete. This function could be used to associate input contexts with the output context. The return value of this function should be returned to the application in the `ProcessData` function.

Example

```
if ( done )
return Complete();
else return Pending();
```

Reject()

`Reject()`

Parameter	Meaning	Default Value	Comments

Return value

An integer that should be passed back to the application unchanged.

Comments

Called when it is decided that the input context does not meet the criteria for being a part of the current transaction. The output context should not be modified before this decision is made. The return value of this function should be returned by the `ProcessData` function.

Example

```
if ( UnknownValue )
return Reject();
```

CHAPTER 14: FIREINSPECTOR-SPECIFIC PRIMITIVES

BitfieldInit()

`BitfieldInit(<bitfield_identifier` *`string`*`>, <title` *`string`*`>)`

Parameter	Meaning	Default Value	Comments
bitfield_identifier *string*	The name of the bitfield data structure		Used to refer to this bitfield data structure in subsequent calls to other primitives.
title *string*			Displays in the title bar of the dialog box.

Return value

None.

Comments

`BitfieldInit` initializes the data structure for keeping field information. This field information can be presented to the user in the form of a dialog box (see "`BitfieldDialog()`" on page 58). The primitives `GetNBits` and `NextNBits` both support optional arguments for appending information to this data stucture (see "`GetNBits` -- Additional parameters" on page 57 and "`NextNBits` -- Additional parameters" on page 58).

Example

See "Example for FireInspector-Specific Primitives" on page 60.

GetNBits -- Additional parameters

`GetNBits` contains two additional, optional arguments in FireInspector to add data to the `BitfieldInit` data structure: `bitfield_identifier` and `bitfield_label`. In FireInspector, `GetNBits` has this structure:

`GetNBits(<bit_source` *`list`* `or` *`raw`*`>, <bit_offset` *`integer`*`>, <bit_count` *`integer`*`>, <bitfield_identifier` *`string`*`>`$_{opt}$`, <bitfield_label` *`string`*`>`$_{opt}$`)`

Use `bitfield_identifier` to refer to a bitfield data structure that has been named in a `BitfieldInit` declaration. Use `bitfield_label` to assign a label to the group of bits being read. For example:

```
GetNBits( data, 0, 1, "Identifier", "Label" );
```

In the example, “Identifier” is the value of `bitfield_identifier` and “Label” is the value of `bitfield_label`.

For a complete description of `GetNBits`, see “`GetNBits()`” on page 41.

Example

See “Example for FireInspector-Specific Primitives” on page 60.

`NextNBits` -- Additional parameters

`NextNBits` contains an additional, optional argument in FireInspector to add data to the `BitfieldInit` data structure: `bitfield_label`. In FireInspector, `NextNBits` has this structure:

`NextNBits(<bit_count` *`integer`*`>, <bitfield_label` *`string`*`>`$_{opt}$`)`

Use `bitfield_label` to assign a label to the bits being read, and put them in the bitfield referred to by the previous call to `GetNBits`. For example:

```
NextNBits( 3, “Label” )
```

For a complete description of `NextNBits`, see “`NextNBits()`” on page 42.

Example

See “Example for FireInspector-Specific Primitives” on page 60.

`BitfieldDialog()`

`BitfieldDialog(<bitfield_identifier` *`string`*`>, <width` *`integer`*`>)`

Parameter	Meaning	Default Value	Comments
bitfield_identifier *string*	The name of the bitfield data structure		Refers to the bitfield data structure from which to get the data.
width *integer*	The width, in bits, of the data field		Currently, width specification of other than 32 bits will yield unpredictable results.

Return value

None.

Comments

`BitfieldDialog` brings up a dialog box in which the user can view the fields that were parsed for a particular protocol-dependent data structure. The primitive returns when the user hits the 'OK' button on the dialog box, which causes the box to close. `BitfieldDialog` is not usually called directly while parsing, but instead is handed off in a menu descriptor when constructing a cell (see "`AddCell` -- Supplementary `additional_info` constants" on page 59).

Example

See "Example for FireInspector-Specific Primitives" on page 60.

`AddCell` -- Supplementary `additional_info` constants

The `AddCell` `additional_info` parameter contains a supplementary, optional, constant value in FireInspector that builds a menu cell: `_MENU`. It's used to create a cell that, when left-clicked upon, will bring up a menu.

The descriptor `_MENU` serves as the head of a list that defines a menu:

```
[_MENU, <menu_name string>, <menu_items list>]
```

The first item, `_MENU`, identifies the list as containing a description of a menu. The second item, `menu_name`, is the name that appears at the top of the menu.

The third item in the list is the `menu_items` list. It can contain one or more items; each of those items is a `menu_item_elements` list. A `menu_item_elements` list is structured as follows:

```
[<menu_action_id integer>, <menu_entry string>,
<menu_action_function string>, <menu_arguments list>]
```

The value of `menu_action_id` is an integer that uniquely identifies a menu command. This should be unique among all of the cells of a transaction.

The value of `menu_entry` is a text string that appears as an entry on the menu. When selected, it brings up the dialog identified by the `menu_arguments` list.

The value of `menu_action_function` is string that identifies the function to be called if the menu entry is selected.

The value of `menu_arguments` is a list of arguments for `menu_action_function`.

```
[<argument_list list>]
```

For a full description of `AddCell`, see "`AddCell()`" on page 45.

Example

See “Example for FireInspector-Specific Primitives” on page 60.

Example for FireInspector-Specific Primitives

Example Code

```
# Currently, BitfieldInit() must be contained within
a CollectData module:

CollectData()
{
   data = '0123456789abcdef';

   BitfieldInit( "stuff", "This is a view field
dialog" );
   GetNBits( data, 0, 1, "stuff", "a" );
   NextNBits( 2, "b" );
   NextNBits( 3, "c" );
   NextNBits( 4, "d" );
   NextNBits( 5, "e" );
   NextNBits( 6, "f" );
   NextNBits( 11, "g" );
   NextNBits( 32, "second quadlet");

data = '89abcdef01234567';

    BitfieldInit( "more_stuff", "This is another view
field dialog" );
    GetNBits( data, 0, 1, "more_stuff", "h" );
    NextNBits( 2, "i" );
    NextNBits( 3, "j" );
    NextNBits( 4, "k" );
    NextNBits( 5, "l" );
    NextNBits( 6, "m" );
    NextNBits( 11, "n" );
    NextNBits( 32, "fourth quadlet");
}

BuildCellList( )
{
   AddCell( "Click here for menu", null, null,
0x0000ff, _MONOFIELD,
```

```
   # _MENU descriptor and menu_name
   [_MENU, "Menu entries appear below:",
      # Begin menu_items
      [
         # Begin first menu_item_elements list
         [
            # menu_action_id
            0,

            # menu_entry
            "View Stuff",

            # menu_action_function
            "BitfieldDialog",

            # Begin menu_arguments list
            [
               # bitfield_identifier
               "stuff",

               # width
                32

            # End menu_arguments list
            ]
         # End first menu_item_elements list
         ],

         # Second menu_item_elements list
         [1, "View More Stuff", "BitfieldDialog",
["more_stuff", 32]]

      # End menu_items
      ]
   # End _MENU descriptor list and AddCell
   ]);
}
```

Example Output

The `AddCell` entry builds the cell that contains the menu:

Figure 14-1: Menu cell

Clicking on the cell brings up the menu defined in the _MENU descriptors list:

Figure 14-2: Menu

Clicking on a menu entry brings up the dialog box:

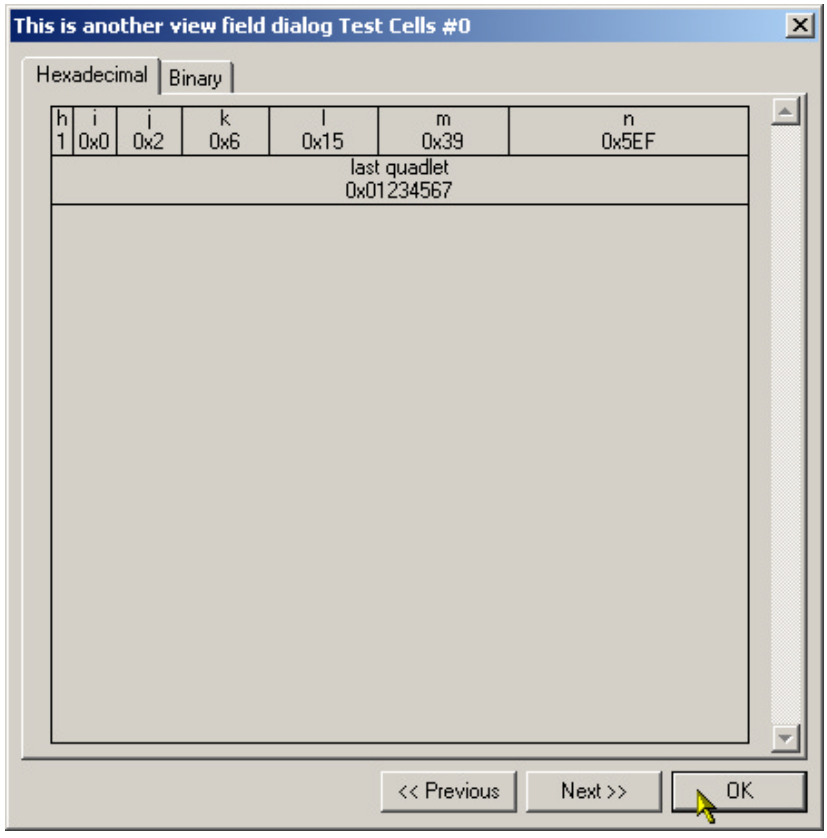

Figure 14-3: Dialog box

The dialog box displays the data that is referred to by the bitfield identifier `“more_stuff”`.

CHAPTER 15: MODULES

Modules are a collection of functions and global data dedicated to decoding a certain type of transaction. Each module consists of one primary file (.dec), and possibly several included files (.inc).

Module Functions

Three functions are used as entry-points into a decoding module. They are called by the application and are used both in the initial transaction decoding phase, and each time that a transaction needs to be displayed.

`ProcessData()`

Called repeatedly with input contexts representing transactions of the specified input types. Decides if input transaction is a member of this transaction, or if it begins a new transaction. This function will be called first using incomplete output transactions. If the input transaction is not accepted into any of the pending transactions, it will be called with an empty output transaction to see if it starts a new transaction.

`CollectData()`

Called with each input transaction that was previously accepted by the function `ProcessData`. Generates all output context data that would be required for input into a higher level transaction.

`BuildCellList()`

Called with the output context generated by the call to `CollectData`, and no input context. This function is responsible for adding display cells based on the data collected by `CollectData`.

Note that there is some flexibility in the use of these functions. For example, if it is easier for a particular protocol to build cells in `CollectData`, cells could be generated there, and `BuildCellList` could be left empty. Another approach would be to have `ProcessData` do everything (generate output data, and build cell lists) and then implement `CollectData` as a pass-thru to `ProcessData`. This will be less efficient in the decoding phase but may reduce some repetition of code. These decisions are dependent on the protocol to be decoded.

Module Data

There are several standard global variables that should be defined in a module which are queried by the application to figure out what the module is supposed to do.

ModuleType

Required. A string describing the role of the script. Currently, only `Transaction Decoder` and `DataBlock Decoder` are valid.

Example

```
set ModuleType = "Transaction Decoder";
```

`Transaction Decoder` uses `ProcessData()`. `DataBlock Decoder` does not.

OutputType

Required. A string label describing the output of the script. Example : `AVC Transaction`

Example

```
set OutputType = "AV/C Transaction";
```

InputType

Required. A string label describing the input to the script. Input and output types should be matched by the application in order to decide which modules to invoke on which contexts.

Example

```
set InputType = "1394 Transaction";
```

LevelName

Optional. A string that names this decoder.

Example

```
set LevelName = "AV/C Test Transactions";
```

DecoderDesc

Optional. A string that describes this decoder. Displays as a toolbar icon tool tip.

Example

```
set DecoderDesc = "View test transactions";
```

Icon

Optional. File name of an icon to display on the toolbar. Must be a 19x19 pixel bitmap file.

Example

```
set Icon = "bitmap.bmp";
```