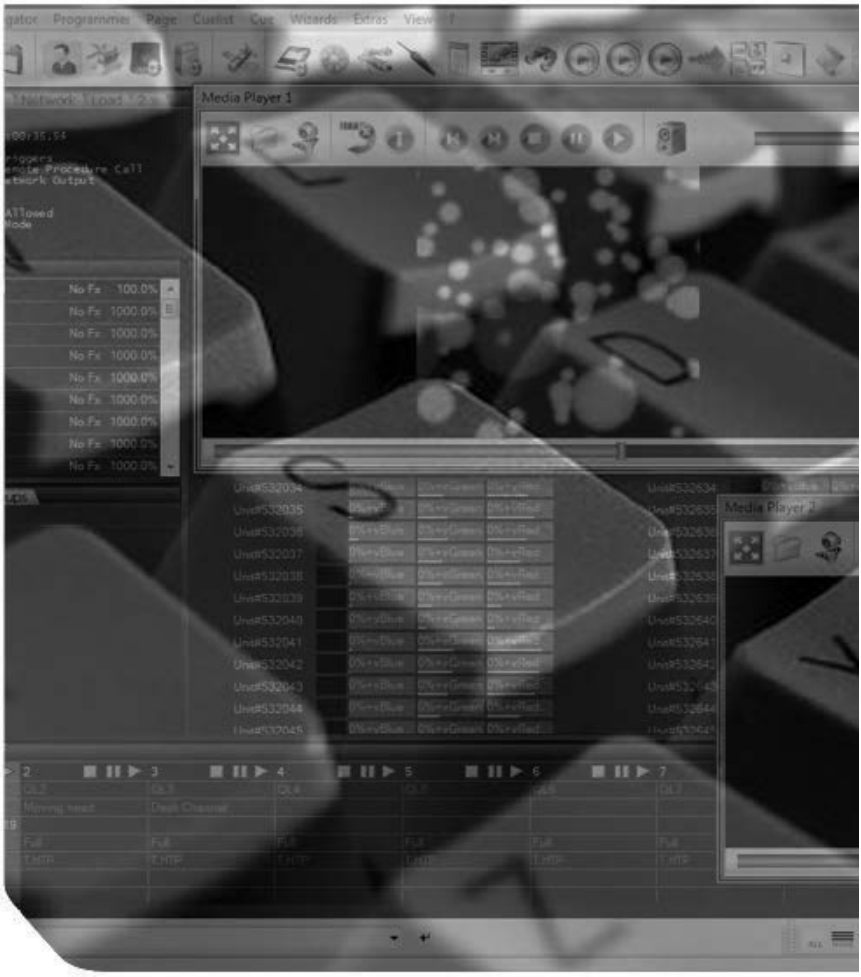
e:cue

LIGHTING CONTROL



Lighting Application Suite 7.0 SR1

Advanced System Manual

Lighting Application Suite 7.0 SR1

Advanced System Manual (original version)

Edition: 10.11.2015

Published by:

OSRAM GmbH
Karl Schurz-Strasse 38
Paderborn, Germany

Available for free download from www.traxontechnologies.com

Contents

1 Changes for LAS 7.0

1.1 New e:script commands

- strToValue: Convert a string to a integer value.
- strToUpper: Convert a string to uppercase.
- strToLower: Convert a string to lowercase.
- strInsert: Inserts a substring at the given index within the string.
- strTokenize: Finds the next token in a target string.
- strDelete: Deletes a character or characters from a string starting with the character at the given index.
- strFindOneOf: Searches a string for the first character that matches any character contained in charSet.
- strLeft: Extracts the leftmost count characters from the string and returns a copy of the extracted substring.
- strRight: Extracts the last (that is, rightmost) count characters from the string object and returns a copy of the extracted substring.
- strRemove: Removes all instances of the specified character from the string
- strSpanIncluding: Extracts characters from the string, starting with the first character, that are in the set of characters identified by charSet.
- strSpanExcluding: Extracts characters from the string, starting with the first character, that are not in the set of characters identified by charSet.
- strTrimRight: Trims trailing characters from the string.
- strTrimLeft: Trims leading characters from the string.
- strTrim: Trims leading and trailing characters from the string.
- strReverse: Reverse a string.
- GetDate: Returns current system date.
- GetTime: Returns current system time.
- DateDiff: The function returns the difference between two dates in days.
- TimeDiff: The function returns the difference between two times in seconds.
- TimeAdd: Adds a specified time interval to a date.
- DateAdd: Adds a specified time interval to a date.
- FormatDateTime: Return the given time as string.
- CuelistGetProperty: Returns the value of the given cuelist property.
- CuelistSetProperty: Sets a new value for the given cuelist property.
- MessageWaitInfo: Present a message box to the user, which closes automatically after a given time.
- MessageWaitWarning: Present a warning message box to the user which closes automatically after a given time.
- MessageWaitError: Present a error message box to the user which closes automatically after a given time.

See the online help for details about the commands.

1.2 Events list updated

The list of events (see „5.5 Event types“) was updated. Some events were missing.

The event VMasterEvent was added.

2 Introduction

The Programmer in e:cue's Lighting Application Suite is a great tool for creating shows for lighting installations, with the integrated tools for automation, remote device management and graphical user interfaces like the Action Pad. Even more powerful it gets with e:script. e:script is a C-like macro language interpreter which makes it possible to realize functions and automation that are custom-built for applications. This manual shows the basic structure of e:script as well as language elements and syntax.

We begin with a programming course for the e:script Macro Language, covering the very basics. If you are already familiar with programming, you might want to skip directly to a later chapter, where e:script is used for several advanced purposes like automation and device access.

The other chapters give many examples for interfacing and also explain the e:net protocol for software developers who want to implement it in their products. In the last chapter, the e:cue Programmer Toolbox is explained along with some other tweaks and tricks for using the e:cue Programmer.

2.1 The e:script Language

The e:script language is a macro language embedded in the e:cue Programmer. Based on the C programming language and equipped with lots of special language commands, the e:script language offers great possibilities for advanced control of the e:cue Programmer. Most of the features, that the e:cue Programmer offers in its graphical user interface, such as cuelist control and fixture manipulation, are accessible via particular e:script commands. This enables advanced users to benefit from the full potential of the e:cue Programmer when creating shows.

This tutorial has been designed to form a starting point for learning to create own e:script macros. We start with the basics of programming, working on bit by bit in order to establish the prerequisites for more complex topics. If you are already experienced in programming, you can skip the first chapters, but for a beginner it is strongly recommended to work through all chapters in order.

2.2 Language features

- Based on ANSI C
- Local and Global Variables: Integer, String, Arrays
- Control Loops: while, for
- Program Control: if, switch/case/break, return, exit
- Nesting
- Arithmetic operations: sqrt, radius, sin, cos, tan, etc…
- Custom user functions
- Convert HSB to RGB values
- Call other macros

Cuelists

- Read/write access to all cuelists
- Start or stop cuelists
- Copy, paste, and delete cuelists
- Change Mutual Exclude cuelist properties

Single Cues

- Read/write access to all cue properties
- Record new cues

Masters

- Read/write access to GrandMaster, Submaster, and Versatile Masters (formerly known as Speed Masters)

Other Programmer Functions

- Read/write access to all fixture channels
- Read/write access to all patch data: Patch new fixtures etc.
- Set preheat values
- Access to Live FX
- Read/write access to e:cue programmer sections (cluster size etc.)
- Access to programmer view: select / deselect fixtures, set values, clear programmer view
- Read backup mode and status of operation
- Read/write access to ActionPad
- Read/write access to DMX start byte for all universes

External Communication

- Read/write access to all e:com terminals and e:cue Terminal Emulators
- Send e:com terminal messages into the network
- Setup remote terminals
- Read faderunit jogdials
- Read DMX input data (from e:cue calypso)
- Read/write access to all MIDI data (note/controller/program or binary)
- Start and stop Timecode
- Read/write access to all RS232 data (string or binary)
- Communicate with eFlashPlayer on remote machines
- Read/write access to all Network Parameters for communication with other applications

Remote Machine Control

- Remote PC shutdown and restart via e:cue Terminal Emulator
- Send WakeOnLAN packets to power up remote PC's

System Functions

- Read the system time and access Sunrise Timer
- Shut down and reboot the system
- Play sounds
- Start external applications via system command interpreter

User Interaction

- Create modal dialogues for user interaction with the following items:
  Images, Text input fields, Value input fields, Password input fields Color Picker input fields, Checkboxes, Combo boxes
- Show system message boxes to the user – OK, OK/CANCEL, YES/NO etc.

2.3 How to read this manual

Code example are set in a monotype font:

```
// --- this is a code example
a = a +1;
printf("Incremented a by 1");
```

Command and path names are set in *italic text*.

Language references are shown in a box:

> MessageOk
> int MessageOk(string text);
>
> Shows a standard message box to the user.

Code examples and snippets

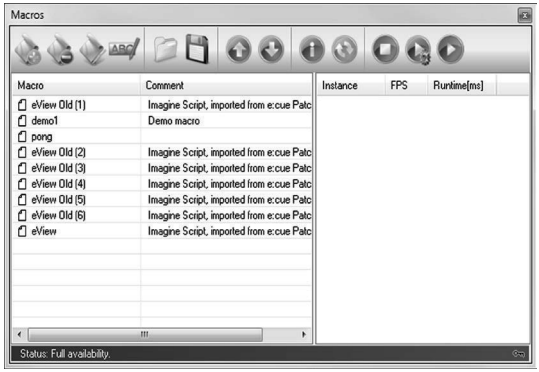
As the examples in this manual is coded in UTF-8, like most PDF text books, the code examples are also available as standard text files. To get these files visit

www.traxontechnologies.com | Downloads | e:cue Software | Support Files

3 About e:script

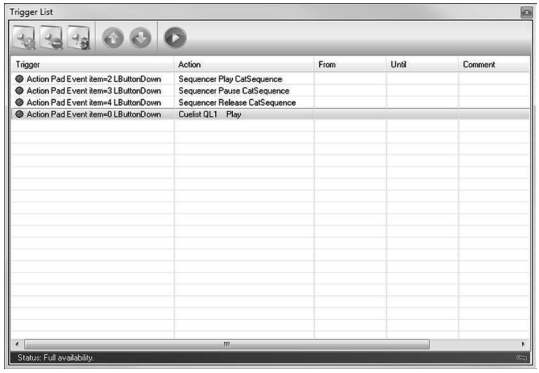
3.1 How to create and run

e:script macros can be used in many places in the Programmer. Use the e:script Macros button on the Main Icon menu, or press Shift+F2 or `View | e:script Macros` from the main menu. Here you can create, edit, delete, and run macros.



Macros are activated by Actions, Actions can be used e. g. for cues or for Triggers. See the chapter about Actions and Triggers for more information on Actions and Triggers.

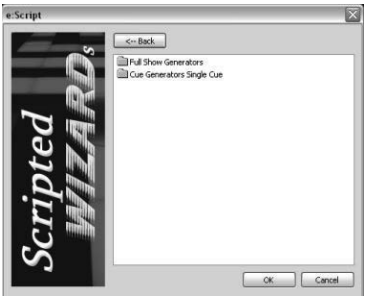
e:script code, building macros, is not meant to realize complete programs or applications like you would do it in C or Java. Macros are executed as one-time “code snippets”: The runtime of a snippet is usually less than 0.01 seconds. To run longer, more “continuous” macros, it is possible to store global values and repeatedly call a macro to run a kind of “state machine”. In this case the macro is called again and again, using persistent variables to store and recall a certain condition or state in the show or in the application.



Usually a macro inside a show would be called either by a Trigger rule or by a Cue Action. An example of the first would be “On a certain time or when an Action Pad button is pressed down, run macro X”.

Another example would be a cuelist with a single cue that calls macro X via a Cue Action. If you want to create a kind of continuously running program, e. g. an animation or a game, you would write a macro that saves its state in global variables and call this macro from an Action in a single cue in a cuelist. Set the cuelist to loop mode, control timing by wait time, fade-in and fade-out timing and the macro is executed again and again. This will be explained in more detail in a later chapter. More information on Automation trigger rules and Cue Actions can be found in the e:cue Programmer system manual.

3.2 e:script Wizards



External e:script macros can be run using the e:script Wizard – available from the main menu via Wizards | e:script Wizard. These macro files need to be put inside a subdirectory of the e:cue Programmer's main directory (typically `C:\Program Files\ecue\Programmer Vx.X\Resources`) named Wizards.

The e:script Wizard will display all files with the correct extension that it finds in that subdirectory as executable macros. To be recognized by the e:script Wizard the files need to have a .cpp extension. You can edit the macro files as plain text, using Windows Notepad or any other standard text or code editor.

How to run the code examples

There are two ways to run the several macro code examples. You can create a new macro and simply copy the desired macro code into the e:script editor. Save the macro and run it by pressing the "Run Macro"-button.

The other way is to create a text file inside the "Wizards"-subdirectory of the e:cue programmer directory. Do not forget to give the file a .cpp extension or the e:script Wizard will not recognize it. Now copy the macro code into the file. Afterwards you should be able to run them straightaway from the e:script Wizard.

This Advanced System Manual is not meant as a general introduction to the programming language C, it is only a language reference. Please ask Google for hits on "c programming language tutorial".

4 Language structure

For all functions, keywords and classes please see `Help | e:script Language Reference` in the Programmer.

4.1 Keywords

In general, e:script language and syntax is the same as for the ANSI C language regarding keyword use, variable naming and writing format. All keywords are case sensitive, so if you want to call the command `CuelistStart`, you have to write it exactly this way. Writing the command like `CueList-Start` will be misinterpreted. The same applies for variable names, so that e.g. the variables number and Number are different. More about variables can be found in the following chapter. Each statement has to be finished with a semicolon.

4.2 Comments

In e:script, a comment begins with the // character pair. The complete line written after the // will be ignored by the interpreter. For example, this is a syntactically correct comment:

```
// This is a comment
```

You can also add a comment after a line of code to describe your purpose:

```
a = 2 - 1;  // Subtract 1 from 2 and store the result in var a
```

The comment form

```
/* This is a comment */
```

from ANSI C is not supported!

4.3 An e:script example: Hello World

This is the code.

```
// This is the "Hello, World" macro
MessageOk("Hello, World!");
```

When you run the macro, you should see the following message box:

The `MessageOk` function is e:script-specific, not available in C.

> MessageOk
> int MessageOk(string text);
>
> Present a standard message box to the user.

`MessageOK` will only accept parameters of the type string. Instead of `MessageOK` you could have also used the `printf` function, which puts the text into the e:cue programmer logbook instead:

```
printf("Hello, World!\n");
```

Note the \n at the end: This signifies to the `printf` command that it has to insert a carriage return and go down to the next line. More about control characters like \n can be found later on.

The e:script language has a very rich set of functions and classes. See the Online e:script reference under Help | e:script Language Reference for an overview. There you will find complete overview of operators and functions, which are out of scope of this introduction.

4.4 Variables

The types of variables offered by e:script are:

`int` - integer: a whole number such as 127.

`int arrayname [size]` – We will be discussing arrays in a further section.

`string` - a string of characters, such as “Hello, World”.

Variables must not use the words that e:script uses for command names, such as `MessageOk` or `StartCuelist`. All variables must begin with a letter (or a “_” character for global variables, but this will be discussed later on).

Integer Variables

An int variable can store a 32-bit whole number (the range is -2.147.483.648 to 2.147.483.647). No fractional part is allowed. To declare an int you use the instruction:

int variablename;

For example:

```
int a;
```

To assign a value to our integer variable we would use the following e:script statement:

```
a = 10;
```

Be aware that the e:script language, like most common programming languages, uses the “=” character for assignment.

String Variables

String variables in e:script keep strings up to a length of 255 encoded in ANSI-standard.

To declare a string you use the instruction:

```
string variable name;
```

For example:

```
string MessageText;
```

To assign a text to our string variable we would use the following e:script statement:

```
MessageText = “Hello, World!”;
```

Since we can pass variables to functions, our “Hello, World” macro could also be written like this:

```
// This is the “Hello, World” macro with a
// passed message text

string MessageText;

MessageText = “Hello, World!”;
MessageOk(MessageText);
```

The `MessageOK` function expects a string type parameter, the following would not work:

```
int a;
a = 10;
MessageOK(a);
```

and you would receive an error message in the e:cue Programmer’s logbook saying “Expression

expected".

```
Here: MessageOK(a);
helloworlddoesntwork, line 5, col 1:Expression expected.
```

4.5 Arithmetic Operations

Addition: c = a + b;
Subtraction: c = a – b;
Multiplication: c = a * b;
Division: c = a / b;

Precedence of arithmetic expressions

e:script support the arithmetic precedence for expressions. So

```
a = 10 + 2 * 5 - 6 / 2;
```

will not evaluate to 27, but to 17. To avoid confusion, use brackets. This will also make the code more legible, e. g.

```
a = 10 + (2 * 5) - (6 / 2);
```

This calculates the same as the example above, but can be conceived much easier.

Variable increment/decrement

The C-like increment/decrement expression is also available:

```
a++;
a--;
```

The increment operator ++ and the equivalent decrement operator --, can be used as either prefix (before the variable) or postfix (after the variable). Note that ++a increments a before using its value; whereas a++ means use the value in a then increment the value stored in a. In most cases there will be no difference in using prefix or postfix operators but sometimes, especially when using the assignment operator = at the same line, something unexpected might happen:

```
b = a++;
```

After this line of code, b stores the "old" value of a, while a has been incremented by 1. If we use the postfix operator instead, we will recognize a difference:

```
b = ++a;
```

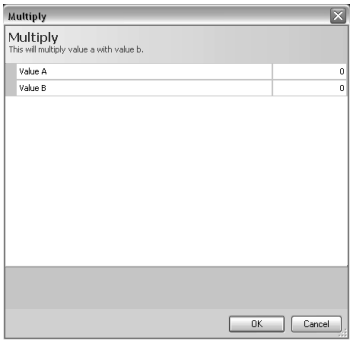
In this case, b and a store the same value, precisely, the incremented value of a.

4.6 Advanced Arithmetics

Command	Description	Remarks
abs(x)	Returns the absolute value of a number.	Returns ‘value’ if value is above or equal zero. ‘-value’ if value is below zero.
atan2(x, y)	Returns the angle of a given point (x, y) relative to the origin (0, 0).	
cos(x)	Returns the cosine of a given number.	Because the radian measure system is not compatible with integer numbers input values must be multiplied by 1000. For the same reason the return value is also multiplied by 1000.
radius(x, y)	Returns the distance of a given point (x, y) to the origin (0, 0).	The radius command calculates sqrt(x*x + y*y).
sin(x)	Returns the sine of a given number.	Because the radian measure system is not compatible with integer numbers input values must be multiplied by 1000. For the same reason the return value is also multiplied by 1000.
sqrt(x)	Returns the square root of a given number.	Only large values for x make sense since the result is integer.
tan(x)	Returns the tangent of a given number.	Because the radian measure system is not compatible with integer numbers input values must be multiplied by 1000. For the same reason the return value is also multiplied by 1000.

4.7 Input Dialogues

e:script supports input and output dialogs to ask for inputs or to display hints or messages. The following code is a short example for input dialogues.




```
int a, b, c;
string myResult;
```

Next, we will write the code that tells the e:cue programmer to display a modal dialogue to the user, asking for input:

```
PropDlgInit("Multiply");
PropDlgSetHeaderText("Multiply\nThis
will multiply value a with value b.");
PropDlgAddInt("Value A", "", a, 0, 0);
PropDlgSetInfo("Enter the value of
A");
PropDlgAddInt("Value B", "", b, 0, 0);
PropDlgSetInfo("Enter the value of B");
PropDlgDoModal();
```

This code produces a modal dialogue with some text in the header, two input boxes for values, A and B, and help texts for the input boxes.

The first command, `PropDlgInit`, declares that we want to create a dialogue. The syntax is as follows:

PropDlgInit
int PropDlgInit(string caption);

Create a new dialog.

`PropDlgSetHeaderText` determines what the dialogue heading textbox.

```
PropDlgSetHeaderText("Multiply\nThis will multiply a with b");
```

The `\n` inserts a line break. In the `PropDlgSetHeaderText` command, any text after \n is the sub-caption and will be written below the Header in smaller text size.

Next, to add fields to enter information, a and b. The `PropDlgAddInt` function takes five parameters, separated by commas:

PropDlgAddInt
int PropDlgAddInt(string caption, string unit, int-ref variable,
int min, int max);
Add an integer value to the dialog, min=0, max=0 disable range checking

The first parameter is the caption of the value. The second parameter is a description of the unit. e.g. “pixels”, “fixtures”, or “terminal number”, to keep the user informed about what they are entering. If you want to add a parameter description, you should add a leading blank to ensure that there is some space between the input field and the description itself.

The third parameter is the name of the variable that we want the Programmer to use for storing whatever was entered. The last two parameters tell the programmer to put a limit on the range of numbers the user can enter, 0 stands for no limit.

Even though some of the parameters are not doing much in the above statement, in programming we always need to pass the right number of parameters when invoking a macro command.

The command `PropDlgSetInfo` always relates to the last inserted element of the dialogue. Therefore, to associate a help text with an item, `PropDlgSetInfo` needs to come straight after the `PropDlgAddInt` statement.

This approach of commands relating to a certain (previously created) element might be a bit confusing at the beginning. However, there are various more commands that work this way, so you should take your time and get a little used to this. You will certainly have to use commands like this more often, especially when creating modal dialogues.

With

```
PropDlgDoModal();
```

we show the dialogue to the user. To complete our macro, refer to the next chapter.

4.8 Using format and printf

The command `MessageOK` only accepts strings as argument. To convert numeric variables to strings use the `format` command, which relates to `sprintf`() in ANSI C.

format
string format(string format, ...);

Returns a string. The format is like at the printf function.

`format` first requires a string, and then an indefinite list of variable parameters.

```
myResult = format("The result of %d multiplied with %d is
%d",a,b,c);
MessageOk(myResult);
```

We could have used `printf` instead and simply put the output into the e:cue programmer logbook (and not use the `myResult` string variable):

```
printf("The result of %d multiplied with %d is %d", a, b, c);
```

The % Format Specifiers for printf and format Commands:

Specifier	Variable type	Display
%d	int	signed integer
%s	string	sequence of chars
%x	int	unsigned hex value

Here the complete example:

```
// Multiply Dialogue
// Multiplies A with B
int a, b, c;
string myResult;

PropDlgInit("Multiply");
PropDlgSetHeaderText("Multiply\nThis will multiply value a with
value b");
PropDlgAddInt("Value A", "", a, 0, 0)
PropDlgSetInfo("Enter the value of A");
PropDlgAddInt("Value B", "", b, 0, 0);
PropDlgSetInfo("Enter the value of B");
PropDlgDoModal();
c = a * b;
myResult = format("The result of %d multiplied with %d is %d", a ,
b, c);
MessageOk(myResult);
```

Hint: You are not limited to using variables as parameters. It is also possible to directly perform arithmetic operations or use commands as parameters for `printf` and `format`.

```
myResult = format ("The result of %d multiplied with %d is %d", a
, b, a * b);
```

4.9 Control Characters

These control characters will only be interpreted using `printf` or `format`. This includes \\ for a backslash and \" for a quotation mark as \ and " are special characters for themselves in e:script (as for \ marking a control character sequence and " as a marker for the beginning and the end of a string). The different control characters can be found in the following table:

Control Character	Description
\n	line feed
\r	carriage return
\\	'\' (backslash)
\"	" (quotation mark)

4.10 Compound statements

For control structures and conditional execution e:script supports compound statements in the C language syntax:

```
{
            statement1;
            statement2;
            statement3;
}
```

Notice that the statements within the curly brackets have to be terminated by semicolons as usual. The closing bracket does not have a semicolon as termination. You can use a compound statement anywhere you can use a single statement.

4.11 Conditional execution: if

The `if` statement has the following form:

```
if (condition)
            statement;
else
            statement;
```

As an example, we can use `MessageYesNo` in an `if` statement and use a statement block instead of a single statement:

```
if (MessageYesNo("Click Yes, please") == 1)
{
            MessageOK("Thank you!");
}
```

Here's an example macro using two `if` statements:

```
if (a < b)
            MessageOK("First number is smaller    than second");
if (b < a)
            MessageOK("Second number is smaller than first");
```

4.12 Relational Operators

To compare two values you can use the standard symbols:

>	(greater than)
<	(less than)
>=	(for greater than or equal to)
<=	(for less than or equal to)
==	(to test for equality)
||	(Boolean OR)
&&	(Boolean AND)

As mentioned before, the reason for using two equal signs for equality is that the single equal sign always means store a value in a variable - i.e. it is the assignment operator.

```
if (a = 10)  // wrong
if (a == 10) // correct
```

The e:script interpreter will generate an error message if it parses an expression like the first one. Just as the equals condition is written differently from what you might expect so the non-equals sign looks a little odd. You write “does not equal” as `!=`. For example:

```
if (a != 0) // is 'if a is not equal to zero'.
```

An example macro showing the `if else` construction now follows:

```
\\ Even if i click Cancel, it prints Cannot divide by zero.
int num1, num2, numresult;

PropDlgInit("Division");
PropDlgSetHeaderText("Whole Number Division\n");
PropDlgAddInt("Enter first number", "", num1, 0, 1000);
PropDlgAddInt("Enter second number", "", num2, 0, 1000);
PropDlgDoModal();

if (num2 == 0) printf("\n\nCannot divide by zero\n");
else printf("\n\nAnswer is %d\n\n",num1/num2);
```

This macro uses an `if` and `else` statement to prevent division by 0 from occurring.

4.13 Boolean values

The e:script language does not have a concept of a Boolean variable line ANSI C. TRUE is represented by any numeric value not equal to 0 and FALSE is represented by 0. This fact is usually well hidden and can be ignored, but it does allow you to write `if (a != 0)` just as `if(a)` because if `a` isn't zero then it also acts as true. You should care about clarity of your of you code if you make us of this. Reading something like `if(!done)` as ‘if not done’ is clear, but `if(!total)` is more or less vague.

4.14 Conditional execution: switch

The `switch` statement is a multiple selection statement. It is used to select one of several alternative paths in macro execution and works like this: A variable is successively tested against a list of integer or character constants. When a match is found, the statement sequence associated with the match is executed. The general form of the `switch` statement is:

```
switch(expression)
{
            case constant1: statement sequence; break;
            case constant2: statement sequence; break;
            case constant3: statement sequence; break;
            default: statement sequence; break;
}
```

Each case is labelled by one, or more, constant expressions (or integer constants). The `default` statement sequence is performed if no matches are found. The `default` statement is optional. If all matches fail and `default` is absent, no action takes place. When a match is found, the statement sequence associated with that `case` is executed until `break` is encountered. The switch statement differs from if in that switch can only test for equality, whereas the if conditional expression can be more complex.

The following example macro lets the user enter two numbers and decide what mathematical operation is wanted to perform with the numbers, using a `ComboBox` dialogue element.

```
int a, b, mathOperation;
mathOperation = -1;
string myText;

PropDlgInit("Calculator");
PropDlgSetHeaderText("Please enter two numbers and select the
            mathematical operation\n");
PropDlgAddInt("Enter first number", "", a, 0, 0);
PropDlgAddComboBox("Select mathematical operation to be
            performed",mathOperation,"Add;Subtract;Multiply;
            Divide");
PropDlgAddInt("Enter second number", "", b, 0, 0);

PropDlgDoModal();

switch(mathOperation)
{
            case 0:
              myText = format("%d plus %d equals %d \n", a, b, a +
b);
              break;
            case 1:
              myText = format("%d minus %d equals %d \n", a, b, a
- b);
              break;
            case 2:
              myText = format("%d multiplied by %d equals %d \n",
a, b,
                          a * b);
              break;
            case 3:
              if (b) myText = format("%d divided by %d equals %d
```

```
\n",
                              a, b, a / b);
            else myText = format("Error: Cannot divide by
zero.");
            break;
          default:
            myText = format("Error: Please select a mathematical
                              operation.");
}
MessageOk(myText);
```

In the macro above, note the `PropDlgAddComboBox` element: The parameters it takes are a caption string, a variable that receives the selection the user took (zero-based, i.e. if the user selects the first item the variable will be zero), and a string which is a list of items – the items being separated by semicolons.

Note also the `if` statement inside the `switch` statement: If the user did select "divide", this makes sure we don't try to divide anything by zero (which would stop the macro with an error message).

Finally, note the "default:" statement inside the `switch` statement: This is the statement that gets executed if the user did not select anything from the ComboBox. This works only because we took care to set `mathOperation` to –1 in the beginning of the macro, and the "default:" statement only gets executed if `mathOperation` does not equal any of the numbers 0, 1, 2, 3.

4.15 Repeated execution: while

You can repeat any statement using the while-loop:

```
while (condition)
          statement;
```

The `condition` is a test to control how long you want the statement to carry on repeating.

```
while (condition)
{
          printf("Hello World");
          printf("\n");
}
```

The `while` checks out the condition, and if it equals `TRUE` the statement is carried out, then the condition is checked again, if it is still `TRUE` the statement is carried out again etc. - If the condition turns out to equal `FALSE`, the statement is not carried out even once and the macro moves on to the next statement.

4.16 Conditions or Logical Expressions

The condition (or Logical Expression), which can return either TRUE or FALSE, can be any test of one value against another. For example:

20 > 10 is always TRUE; the while loop would keep running forever.
20 > 50 is always FALSE, the while loop would never run once.
a > 0 is only TRUE if `a` contains a value greater than zero;
b < 0 is only TRUE if `b` contains a value less than zero.
a == 10 is only TRUE if `a` equals the value 10.

Note that the test for 'something equals something else' uses the character sequence == and not =. Wrong use of the double or single equal sign to have different meanings is a cause of many a

program bug for beginner and expert alike!

Example:

```
int count;
count = 0;
while (count < 100)
{
            ++count;
            printf("Hello World!\n");
}
```

4.17 MessageYesNo in a while loop

`MessageYesNo` displays a message box with "Yes" and "No" buttons to the user. The function returns `TRUE` if the user has clicked "Yes" and `FALSE` if the user has clicked "No". We can use `MessageYesNo` as a condition inside a while-loop. As long as the user clicks "Yes", the while loop is executed. Note that because the function call is inside the `()` brackets, there is no semicolon after `MessageYesNo`.

```
int counter;
counter = 0;

while (MessageYesNo("Keep going?"))
{
            counter++;
}

string message;

message=format("You clicked YES %d times before you got bored.",
counter);
MessageOk(message);
```

Let us make this example a bit more complex: This time we set a maximum number of times the user can click "Yes".

```
int a, maximum, keepGoingFlag;
a = 0;
maximum = 10;
keepGoingFlag = 1;
string message;

while (keepGoingFlag == 1)
{
            message = format("You've clicked YES %d times.\n
                                    Click YES
            to continue", a);
            if(MessageYesNo(message) && a < maximum)
                        a++;
            else
                        keepGoingFlag = 0;
}
```

```
if (a==maximum)
            message=format("You clicked YES %d times, that's
enough!",
            a);
else
            message=format("You clicked YES %d times before you
            clicked NO.", a);
MessageOk(message);
```

We use a signal variable, `keepGoingFlag`, as a condition for the while loop. Then we use `MessageYesNo` and a counter, `a`, as a condition for the `if` statement. As long as the user clicks YES and the maximum isn't reached , `keepGoingFlag` is left untouched, which keeps the loop running, and the counter a is simply increased. But as soon as either one of the conditions becomes false (i.e. the user clicked NO or the maximum was reached), the `else` statement is executed: `keepGoingFlag` is set to zero which stops the while loop. The user gets a different message depending on a simple check wether the loop stopped because the maximum was reached or the user clicked NO.

4.18 Repeated execution: for

The `for`-loop repeats a set of instructions while a certain condition is true.

```
for (counter = start_value; counter <= finish_value; ++counter)
            statement
```

Example: To print the numbers from 1 to 100, you could use:

```
for (i = 1; i <= 100; ++i)
            printf("%d \n", i);
```

Here is one example that stops the first 64 cuelists:

```
int i;
i = 0;
for (i = 0; i < 64; ++i)
            StopCuelist(i);
```

Note how the variable `i` is initialized with zero in this example. This is because the parameter for `StopCuelist` has to be zero-based.

4.19 Nesting

All conditional and repeating commands can be nested in e:script.

Example: Think of a number

```
target = random(100);
```

will store a random number between 0 and 99 in the integer variable `target`.

```
int target, guess;
string myText;
myText="Your first guess:";

while(MessageYesNo("Do you want to guess a number?"))
{
            target = random(100);
            while(target!= guess)
```

```
            {
                        PropDlgInit("Guess the Number");
                        PropDlgSetHeaderText("Guess the number
between 0 and
                         100.\nPress 'Cancel' to stop.");
                        PropDlgAddInt(myText, "", guess, 0,100);
                        if(!PropDlgDoModal())
                        {
                         MessageOk("Canceled");
                         exit();
                        }
                        if (target > guess)
                         myText = "Too low! Guess again:";
                        else
                         myText = "Too high! Guess again:";
            }
            MessageOk("\n Well done! You got it! \n");}
```

Note the following line:

```
if(!PropDlgDoModal()){...}
```

This does two things at once. First, it displays the dialogue by calling `PropDlgDoModal`. Secondly, if the user clicks "Cancel", `PropDlgDoModal` returns FALSE, which leads to the statement in brackets `{}` is executed: With the `!` character we are checking for a FALSE condition.

Example: A Pocket Calculator with a while-loop

```
int a, tempResult, mathOperation, keepGoingFlag;
mathOperation = -1;
keepGoingFlag = 1;
string myText, operationText;

operationText = "Enter first number";

while (keepGoingFlag)
{
            PropDlgInit("Calculator");
            PropDlgSetHeaderText("Please enter a number and select
                        the mathematical operation\nPress 'Can-
cel' to stop.");
            PropDlgAddComboBox("Select mathematical operation to
be
                        performed",mathOperation,
                        "Add;Subtract;Multiply by;Divide by");
            PropDlgAddInt(operationText, "", a, 0,0);
```

```
            if(!PropDlgDoModal())
            {
                        keepGoingFlag=0;
                        myText=format("Canceled-Final Result is
%d",tempResult);
            }
            switch(mathOperation)
            {
                        case 0:
                        tempResult = tempResult+a;
                        operationText = format("%d (last
                                operation:Add)",tempResult);
                        break;
                        case 1:
                        tempResult = tempResult-a;
                        operationText = format("%d
                                (last operation:Subtract)",tempR
esult);
                        break;
                        case 2:
                        tempResult = tempResult*a;
                        operationText = format("%d
                                (last operation:Multiply)",tempR
esult);
                        break;
                        case 3:
                        if (a)
                        {
                                tempResult = tempResult/a;
                                operationText=format("%d
                                (last
operation:Divide)",tempResult);
                        }
                        else
                                operationText=format("Temp re-
sult: %d
                                   (Cannot divide by
zero)",tempResult);
                        break;
                        default:
                        operationText=format("Enter first num-
ber:");
            }
}
MessageOk(myText);
```

4.20 Global variables

If a variable name starts with an underline "_", it will be declared as 'global', that means it persists even when the snippet has ended. This is a great advantage if you want to keep track about the state of something longer than the usual quite short runtime of a e:script macro. Simply declare

the variable with an underline:

```
int _myGlobalInteger;
```

Unfortunately, there is a peculiarity in it, what makes handling of global variables a bit tricky: If you start a snippet that created a global variable a second time, you will get an error when the macro reaches the line where the variable was declared the first time. This is because you cannot declare the same variable more than once. To handle this situation, you have to query the existence of a variable and, if it already exists, skip its redeclaration. The function `undefined` returns true if the given variable does not exist yet. Example:

```
if (undefined(_hello))
            int _hello;
```

That way only the first time that the macro runs, `_hello` will be declared.

If you want to clear the global variables, the  button in the `Triggers->Macros` window lets you delete all global variables.

To find it, simply press T to get into automation and click the „Macros“ tab. You can also use the macro command „clear“ instead.

4.21 Arrays

Arrays in e:script are declared as:

```
vartype name[<variable count>];
```

You have to declare an array before you use it - in the same way you have to declare any kind of variable. Let us see what we have to do to declare an array: For example,

```
int a[5];
```

declares an array called `a` with five elements. Please note: The first element is `a[0]` and the last `a[4]` because e:script, just like C, starts counting at zero.

Of course you can also declare arrays as global with an underscore “_” just like you would do with standard integer and string variables. Using an array, the problem of reading in and printing out a set of values in reverse order becomes simple:

```
int a[5];
int i;

for(i = 0; i < 5; i++)
{
            a[i] = i + 1;
}

for(i = 4; i >= 0; i--)
            printf("%d \n", a[i]);
```

4.22 String Operations

The simplest way to combine two strings is to just use the + operator to “add” them.

```
string a, b, c;

a = "part one ";
```

```
b = "part two\n";
c = a + b;   // this attaches both strings one to another
printf(c);
```

This code produces the output “part one part two”. As you can see, in variable `c`, the combination of `a` and `b` is stored. You can also combine more than two strings and there is no need to use variables. Therefore, it is possible to use a statement like this:

```
a = "Test " + b + "\n";
```

This combines string “Test “, variable `b` and string “\n” and stores the result in variable `a`. It is not possible to attach integers with strings this way, so if you want to do this, keep using the `format` command.

String Comparison

To compare two strings `a`, `b` we use `strcmp` or `stricmp`.

> strcmp
> int strcmp(string txt1,string txt2);
>
> Compare two strings. The comparison is case sensitive.
>
> The return indicates the lexicographic relation of string1 to string2.
> <0 string1 less than string2
> 0 string1 identical to string2
> >0 string1 greater than string2

Here is an example:

```
string a, b;
int i;

a = "a";
b = "b";
i = strcmp(a, b);
printf(" Result is %d \n", i);
```

The result of the string comparison performed will be less than zero, just as we would have expected. But if we have different strings, the result might differ from what you might think of.

Example:

```
string a, b;
int i;

a = "a";
b = "A";
i = strcmp(a, b);
printf(" Result is %d \n", i);
```

Here, you might think that `strcmp` delivers a value less than zero because “A” is a capital letter. But this is not how it works. `strcmp` compares the ASCII codes of the single chars from left to right and if it finds a difference, it will return a value greater or less then zero. While “A” has ASCII code 65, “a” has code 97, so the result stored in `i` will be greater than zero.

The command `stricmp` compares strings the same way but is not case sensitive, so if you e.g. compare two strings “ECUE” and “ecue”, the result will be zero as `stricmp` will treat lowercase and uppercase the same.

4.23 String Intersection and Conversion

To cut a string into pieces, we will make use of the `midstr` command.

midstr
string midstr(string src, int pos, int count);

Return a substring of another string.

Parameters
src - source string
pos - Zero based start position of requested sub string.
count - The length of the requested sub string

One important problem where it is necessary to get a partial string is when you receive input from the serial port and you want to process the data. Let us assume we have a device connected to COM1 that delivers status information about a set of LED matrices which you want to display. You can retrieve a string coming from a serial interface using the command `GetSerialInput` (More about serial IO will be discussed in a later chapter. For this example, the only thing you have to know is that `GetSerialInput` will deliver a string). For the following code example, we arrange a simple pre-made string to simulate a serial input:

```
string serialInput, info;
string t1, t2, t3, t4;

serialInput = "M1:40;M2:32;M3:39;M4:41;"
            // temperature of M1 - M4 in °C
t1 = midstr(serialInput, 3, 2);
t2 = midstr(serialInput, 9, 2);
t3 = midstr(serialInput, 15, 2);
t4 = midstr(serialInput, 21, 2);
printf("Temps: %d %d %d %d \n", t1, t2, t3, t4);
```

With the command `getchar`, we can get the ASCII code of a single char. A colon's ASCII code is 58 while a semicolon has code 59.

We will also make use of the command `strlen`, which delivers the length of a given string.

```
string serialInput;
int i, length, templen;
templen = 0;
serialInput = "M1:40;M2:32;M3:39;M4:41;"  // temperature of M1 -
M4 in °C

length = strlen(serialInput);
for (i = 0; i < length; i++)
{
            if (getchar(serialInput, i) == 58) // did we find a
":"?
            {
                        i++;
                        while(getchar(serialInput, i) != 59) //
run till ";"
                        {
                         templen++;   // count the length inbe-
```

```
tween
                                i++;
                                 }
            }

            if (templen > 0)
            {
                                printf(midstr(serialInput, i - templen,
templen));
                                templen = 0;
            }
}
```

The macro uses `getchar` to look at every char inside the string (that is what the for-loop is for). If it finds one, the while-loop will run, looking for the next semicolon and counting the chars in between. When a semicolon is found, the while-loop will terminate.

At last, the variable `templen` is checked. If it is greater than zero, this indicates that we have found a value. This value is simply print out using the `midstr` command inside a `printf` and the variable `templen` is reset to zero.

The command `val` converts a string to an integer value.

```
string serialInput;
int i, length, templen, processedtemps, threshold;
int temperatures[4];
templen = 0;
processedtemps = 0;
threshold = 55;

serialInput = "M1:40;M2:32;M3:39;M4:41;"  // temperature in °C
length = strlen(serialInput);


for (i = 0; i < length; i++)
{
            if (getchar(serialInput, i) == 58) // did we find a
":"?
            {
                                i++;
                                while (getchar(serialInput, i) != 59) //
run till a ";" is found
                                {
                                 templen++;        // count the length
in-between
                                 i++;
                                }
            }
            if (templen > 0)
            {
                                temperatures[processedtemps] =
val(midstr(serialInput,
```

```
                                                    i - templen, tem-
plen));
                                templen = 0;
                                processedtemps++;
                }

}

for (i = 0; i <4; i++)
{
                if (temperatures[i] >= threshold)
                                alert("Temperature tile %d too high!
Should be less than %d, actual value is %d!\n", i, threshold,
temperatures[i]);
}
```

This macro might be executed every time when the Programmer receives serial input using a trigger. This would allow a continuous temperature checking. Needless to say that in practical use, you will have to implement your own string parsing depending on the input you receive.

We used `getchar` to retrieve the ASCII code of an arbitrary char inside a string. Using `setchar or setuchar`, we can do it the opposite way: we can set any char in a string to the given ASCII value.

4.24 Bitwise Operations

Just like C, the e:script macro language offers various possibilities to access or manipulate single bits of an integer variable. Though this will not be needed in most cases, it can be a very practical way to shorten your code or to make things more efficient.

The e:script language offers several possibilities of direct bit access. The basic binary operators are as follows:

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	bitwise left shift
>>	bitwise right shift

We will first take a look at the logical operators. You already know about the equivalent operators when comparing variables. They compare the particular variables (these have to be of type integer) bit by bit and combine the single results of the bitwise comparisons.

Operation	Result
110 & 011	010
110 | 011	111
110 ^ 011	101

In addition, we have two bitwise shifting operations. Both are used in the following way: You first write the variable or number that you want to shift, and then write the desired shifting operator followed by a number that indicates by how many bits you want to shift. With both operators, the bits shifted out of bounds will be dropped, while the released bit positions will have a 0 assigned.

Example: 10 >> 2 will give out 2 as result. This is because the binary representation of 10 is 1010

and this shifted by 2 bits to the right is a binary 10, which is a 2. Accordingly, the term 10 << 2 will result 40 because 1010 shifted 2 bits to the right is 101000 in binary, which is a decimal 40.

Example

You can combine logical operations with shifting to access single bits in an elegant way: A simple way is to shift the bit that you want to access to the rightmost digit and afterwards perform a bitwise AND with 1. This will cut off the bits left of the certain digit, leaving only the information you want to have. The following code shows this principle by printing out the eight lowest bits of a variable:

```
int i, a;

a = 151; // binary representation: 1001 0111
for (i = 0; i < 8; i++)
            printf("Bit number %d is set to %d.\n", i, (a >> i) &
1);
```

This code only prints out the bits one after another, beginning with the rightmost bit. Instead, you could of course save the variables for further processing.

It is also possible to do it the other way round; i.e. to build up a bit string. This might be needed if you have a device connected that interprets data bit by bit so that you have to convert the information to match the device's requirements. Note that performing left shifts make it easier because you start with the lowest digit. This way the only thing to do is to shift by one bit and then simply add 1 to the shifted integer if you want to set the rightmost bit to 1. Let us take a look at the code to make things clear:

```
int i, bits;
int  output[8];  // will store information for our bitstring
bits = 0;

output[0] = 1;
output[1] = 0;
output[2] = 0;
output[3] = 1;
output[4] = 0;
output[5] = 1;
output[6] = 1;
output[7] = 1;
for (i = 0; i < 8; i++)
{
            bits = bits << 1;
            bits += output[i];
}
printf("Variable 'bits' now holds the value %d.\n", bits);
```

This works very simple: we have an array of eight Numbers where each element holds the value that one of the bits of our output variable named `bits` shall get. In the for-loop, first a shift by one bit to the left is performed. Then simply the value the array `output` holds for the current bit position is added to the variable. The most important thing to pay attention to is to arrange the bits in correct order. The leftmost bit has to be inserted first, the rightmost last (assuming that the bits are put in from the right side, what is the only way that really makes sense here).

Now you have got to know supposable the most important uses of bitwise operations related to the e:script macro language. Here, using these operations makes a lot of sense and shortens your code significantly. You might encounter various other circumstances where bitwise operations might be very useful, some very easy, some a lot more complicated. The following section will give a short glimpse of possibilities you have when working with bitwise operations. This is an example of a more complex use of these.

Multiplication via Bit Shifting

It is possible to implement a multiplication of two integers using only some bit shifting, a few additions and comparisons. This is an algorithm called Ancient Egyption multiplication, which was invented in ancient Egypt and represents a fast method to perform a multiplication in computer hardware when a chip has no dedicated multiplication units. The variables a and b will hold the two factors while the variable c will receive the result.

```
int a, b, c;

c = 0;
a = 2;
b = 10;

while (b != 0)
{
            if ((b & 1) != 0)
                        c += a;
            a = a << 1;
            b = b >> 1;
}
printf("The result is %d.\n", c);
```

While the implementation is of no real use in e:script, it often comes into place when doing machine intimate programming, in most cases using assembler languages.

4.25 Functions

In order to write a function, the first thing to do is write “function” followed by an appropriate name and two brackets `()`. This way the script interpreter will know that the following (compound) statement forms a function with the given name.

At this place, the code will not be executed as this is just a definition for your function.

To give an example, we will create a simple function that will display some text using `printf`:

```
function helloworld()
{
            printf("Hello world!\n");
            printf("This is the hello world e:script using a
function.\n");}
```

This is a very simple example, so we do not have much code here. To call our function, we simply write

```
helloworld();
```

to execute it one time. Like all e:script commands, our function names are case sensitive, so there would be a difference between `helloworld` and `HelloWorld`.

Local function variables

Variables declared inside a function definition as well as parameter variables are local to the function and not visible outside. These circumstances can be very tricky, especially because it is also possible to have variable with the same name inside the function as well as outside.

```
int x;

x = 1;
function intprinter(int x)
{
            printf ("%d\n", x);
}
printf("%d\n", x);
intprinter(5);
printf("%d\n", x);
```

This code snippet will display 1 followed by 5, then 1 again. This might be very confusing, so the best thing to do is to always use variable names inside a function that differ from the names outside.

Return Values

You can return either a constant number or a variable with some value. Getting back this return value works similar to standard macro commands. A statement like

```
x = myfunc();
```

There is only one restriction: You are limited to return values of type int. You can work around this restriction by using a “dedicated return variable”: Just specify a string type variable as to hold the result.

Parameters

The syntax is nearly the same as before, you only have to add a parameter list inside the brackets behind your function name.

Note that you are limited to integer and string type variables, no arrays are allowed. Of course, you can pass single array elements on function call. The following example implements a “maximum”-function with three parameters x, y and z. It will return the maximum over these three values.

```
int max;
function maximum(int x, int y, int z)
{
            if (x > y)
                        if (x > z)
                         return x;
                        else
                         return z;
            else
                        if (y > z)
                         return y;
                        else
                         return z;
}
```

```
max = maximum(12, 11, 7);
printf ("%d\n", max);

max = maximum(34, 0, 2);
printf ("%d\n", max);
```

4.26 Recursions

The basic idea behind recursions is to let a function call itself again. e:script supports recursion.

```
int m, n;

function exp(int b, int x)
{
            if (x > 0)
                        return b * exp(b, x - 1);
            else
                        return 1;
}

m = 2;
n = 4;
printf("The result of %d^%d is %d\n", m, n, exp(m, n));
```

As you can see, exp calls itself with the exponent reduced by one, and multiplies the return value with the base b. The break condition is when the exponent reaches zero.

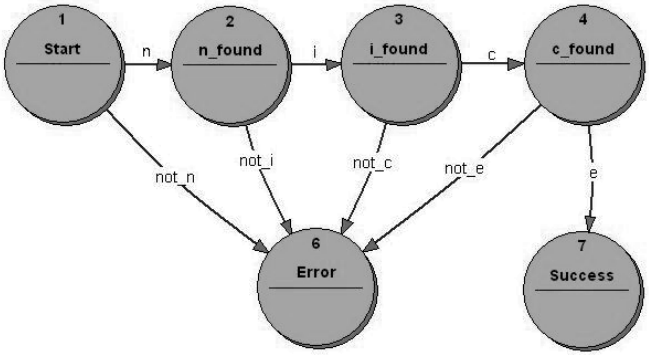
5 State machines

As said before, the concept of an e:script macro is to be a small code snippet with a short runtime. It is possible to create a more complex macro with a long runtime, but first of all the macro language was not designed for that (you have to adjust the build-in runtime limit using `SetRuntimeLimit` manually. Do it on your own risk).

We can easily avoid this problem using the concept of state machines. First, we will explain what this concept means and how we implement it in the e:cue Programmer.

5.1 Basic concepts of state machines

A state machine is an automaton that consists of different so-called states. When the machine receives input, it reacts depending on its current state and depending on the definition, moving into another.



This is an example of a state machine that receives single characters as input. In detail, the machine starts at state one, waiting for input. If it receives an n, it moves on to state two to wait again. (if it receives a character that differs from n, it will move to state six, a kind of “error-state”, and halt. In state two, an n would not help us because now, our machine wants an i to move on to the next non-error-state. This whole process goes on and on until the machine reaches either state six (error) or state seven (success).

5.2 State machines in the e:cue Programmer

The concept of state machines may sound really complicated and theoretical, but it is not. Basically, we will declare some global variables. The current assignments they have represent a certain state. We can attach a macro to a cue so that it will run repeatedly when we start the cuelist. The changes the macro will make to the global variables while its calculation represent different state transitions. This is because changing a value of a global variable will make the macro react differently on its next run.

The only thing to understand is that we will use global variables to record what has happened before (the “state”) and that we are calling a macro repeatedly that will work with these variables (transitions). An example will make things clearer, so have a look at the following section.

5.3 Example: Pong

This is a “Pong”-like macro that makes it appear as if a ball is moving around in a matrix and bouncing off the walls. The strategy is to create a kind of state machine, with global variables forming the states. By calling the macro repeatedly using a cuelist, we simulate state transitions.

```
if (undefined(_pong))
{
            int _pong;
            int _bx, _by, _sx, _sy;
            int _width;
            int _height;
            _width = 32;
            _height = 24;
            _bx = _width/2;
            _by = _height/2;
            _sx = 1;
            _sy = 1;
}

int nx, ny, n;

nx = _bx + _sx;
ny =_by + _sy;

if (ny < 0)
{
            ny = 0;
            _sy = 1;
}

if (ny >= _height)
{
            ny = _height-1;
            _sy = -1;
}

if (nx < 0)
{
            nx = 0;
            _sx = 1;
}

if (nx >= _width)
{
            nx = _width - 1;
            _sx = -1;
}

proClear();
```

```
n = nx * _height + ny + 1;
printf("Activating Unit #%d \n", n);

proSelectOut();
proSelectSingle(0, n);
ClearFx();
SetPosition(0xffff);
proLoadValue(0);
proLoadValue(1);
proLoadValue(2);

_bx = nx;
_by = ny;
```

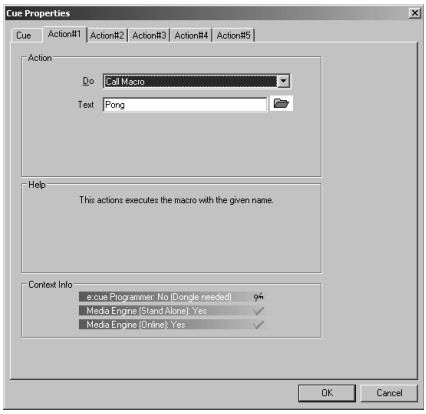
If the global variable _pong is undefined, all the code in brackets is executed, declaring global variables to hold position and speed of the ball, width and height of the matrix. In addition, the ball is put into the middle of the field with initial speed set to 1 on both the horizontal and the vertical axis. Positive values will let the ball move down/right, while negative values will move the ball up/left.

After initialization, the new position of the ball is calculated by adding the speed values to the old position. If the new position exceeds the size of the matrix (that is if a coordinate is below zero or above width/height) the particular speed values will be inverted. That way, the ball will "bounce off the wall".

Then, `proClear` clears the programmer view to get rid of the last picture and the new ball position `n` inside the matrix is calculated. The corresponding fixture is then selected and activated using `proSelectSingle`, `ClearFx` (because we do not want any effect but just a plain white pixel), `SetPosition` and `proLoadValue`. Note that the value is set using hexadecimal format – 0xFFFF = 65535.

Finally, the global variables get updated with the values calculated in this round. The last thing to do is to configure the programmer so that the macro will be called repeatedly:

- Record an empty cue in the first cuelist. Edit the cue properties for that cue.
- First, set the timing to WAIT and use a time of about 0.1 seconds. This is how often the macro is going to be executed, and you can change that time according to how fast you would like your ball to move.
- Click on the "Action#1" tab and select "Call Macro" as an Action.
- Put "Pong" (or whatever name you gave to your macro) as a parameter name, as in the example below:



5.4 Event-driven macro execution

Event-driven macro execution allows macros to turn hibernate and be activated automatically when a particular event occurs. The event triggers a macro function. When this function has run through, the macro will sleep again until the event occurs again. This way the macro will never stop running until explicitely shut down and therefore remember not forget the particular values that the variables hold. Unlike global variables, these variables will only be known inside the macro. This avoids conflicts between different macros.

To create an event-driven macro, perform the following steps:

- Create a macro inside the macro window.
- Inside the macro, register an event that you want to react to, like e.g. When a key on a certain e:bus device has been pressed. Through registration, the event also is connected to a macro function. The corresponding macro code will look as follows:
  `RegisterEvent(EbusKeyPress, OnEbusKey);`
- The first parameter is the event that the macro shall react on, the second parameter is the name of the function that shall be called when the event occurs. The name of that function can be chosen freely. Both names must not be put in quotation marks!
- Create the function itself. Use the name that you chose in the `RegisterEvent-command`. The most important thing is that the function has to have the correct parameter count and types. Both must correspond with the event type that was chosen. In this case, the function needs to have two parameters: an integer for the driver handle of the e:bus device and another integer for the ID of the key that was pressed. The function declaration will look as follows:
  function OnEbusKey(int nDriverHandle, int nKeyId)
  {
  ...
  }
  Only variables that are declared outside the function declaration are persistent through event-triggerings. Variables that are declared inside the function are only visible inside the function and only have a lifetime from declaration to end of the function.
- Design the function to handle the event in a way that you want.

5.5 Event types

The following list shows all different events that can be registered:

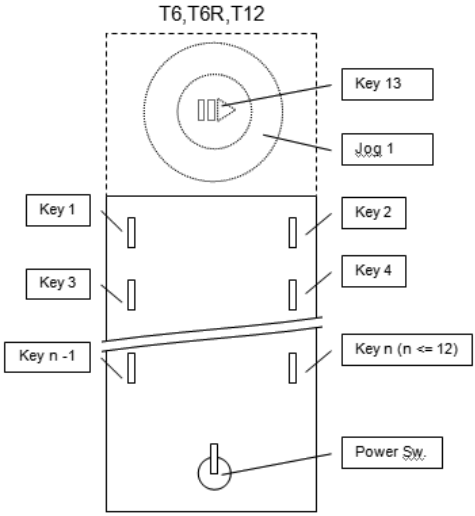
Event Name	Occurance	Parameters
Initialization	When the Programmer has been started & when a new show was loaded.	int p0, int p1, int p2 p0 = 0: Show was loaded. p0 = 1: Reset occurred. p1, p2: not used.
Periodic	Every Minute.	int p0, int p1, int p2 p0: day number (Sunday = 0 … Saturday = 6) p1: hour p2: minute
Hourly	Every Second.	int p0, int p1, int p2 p0: hour p1: minute p2: second
EverySecond	Every Second(same as Hourly).	int p0, int p1, int p2 p0: hour p1: minute p2: second
Cuelist	When a cuelist has been started or stopped; when a cuelist has been idle for some time.	int p0, int p1, int p2 p0: cuelist number p1 = 0: stop event p1 = 1: start event p1 = 2: idle event p2: not used.
VMasterEvent	If a VMaster has been changed.	int p0 p0: V-Master index
Faderunit	When a Faderunit key has been pressed.	int p0, int p1, int p2 p0: always 0 p1: always 1 p2 = key number that was pressed.
MidiNoteOn	When a MIDI note on is received.	int p0, int p1, int p2 p0: channel number p1: note number p2: velocity
TimecodeEvent	When timecode starts or stops.	int p0, int p1, int p2 p0 = 0: timecode starts p0 = 1: timecode stops p1, p2: not used.

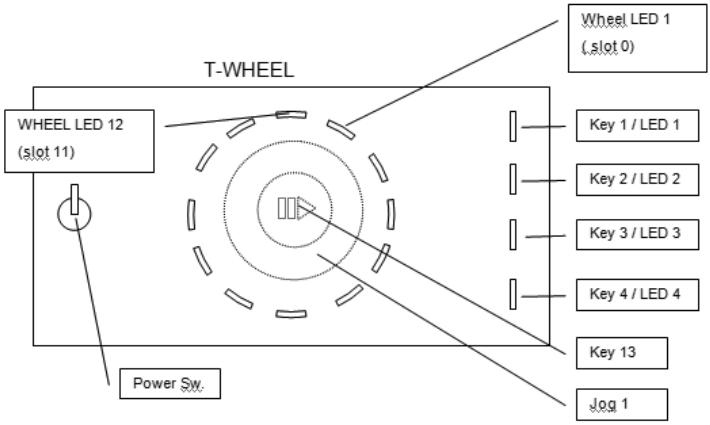
Label	When a label event is triggered.	int p0, int p1, int p2 p0: label ID p1, p2: not used.
TerminalEvent	When a terminal event is received.	int p0, int p1, int p2 p0: serial port number p1:
SerialPort	When a message on a serial port is received. Only works when the first byte is an ASCII char, the second is interpreted as a number.	int p0, int p1, int p2 p0: serial port number p1: ASCII character p2: number
MidiNoteOff	When a MIDI note off is received.	int p0, int p1, int p2 p0: channel number p1: note number p2: velocity
MidiControl-Change	When a MIDI control change is received.	int p0, int p1, int p2 p0: channel number p1: controller p2: value
MidiProgram-Change	When a MIDI program change is received.	int p0, int p1, int p2 p0: channel number p1: program number
ActionPadEvent	When a mouse click is performed, mouse button released or dragging is performed.	int p0, int p1, int p2 p0 – on MButton down, up, drag: ID of the item that is affected. -1 if no item is involved. p1= 0: Mbutton down p1= 1: Mbutton up p1= 2: drag p2 - on Mbutton down, up, drag: cursor position, binary encoded (first 2 byte: Pos Y, last 2 byte: Pos X).
Frame	On every frame (about 30 frames per second)	-
UdpReceive	When a UDP message is retrieved by the generic UDP driver.	int nDriverHandle nDriverHandle: Handle of the UDP driver.
TcpReceive	When a TCP message is retrieved by the generic TCP driver.	int nDriverHandle nDriverHandle: Handle of the TCP driver.
HTTPResponseEvent	If a HTTP response is received.	int p0 p0: Response handle

RDMEvent	If a RDM event is received.	int p0, int p1, int p2 p0: Controller index p1: Port index p2: Event
p0: Controller index	When an e:bus device key is pressed.	int nDriverhandle, int nKeyId nDriverhandle: Handle of the e:bus driver. nKeyId: Key number of the e:bus device.
p1: Port index	When an e:bus device key is released.	int nDriverhandle, int nKeyId nDriverhandle: Handle of the e:bus driver. nKeyId: Key number of the e:bus device.
p2: Event	When an e:bus device jog is pressed.	int nDriverhandle, int nJogId, int nPos nDriverhandle: Handle of the e:bus driver. nJogId: ID of the Jog. nPos: position where the Jog was pressed.
EbusJogRelease	When an e:bus device jog is released.	int nDriverhandle, int nJogId, int nPos nDriverhandle: Handle of the e:bus driver. nJogId: ID of the Jog. nPos: position where the Jog was released.
EbusJogMove	When an e:bus device jog is moved.	int nDriverhandle, int nJogId, int Delta nDriverhandle: Handle of the e:bus driver. nJogId: ID of the Jog. Delta: How far the move was performed.
PeerEvent	If a peer event is received.	int p0, int p1, int p2 p0: Peer ID p1: Event Class p2: Parameter
EmotionFXConnected	If a EmotionFX has been connected to a Sequence	string sSequenceName sSequenceName: the sequence name

6 Key mapping of e:bus devices

The following drawings show the key mapping of the different e:bus devices.





When receiving an e:bus device event, these key IDs are delivered to the function that has been called by the event.

6.1 Example

Assume that we want a single Fixture to light up when a MIDI note on event is received from a connected keyboard, and to darken the fixture again when the subsequent note off event occurs. We can put both events into one macro file to keep everything clear and simple.

First, both events must be registered inside the macro.

```
RegisterEvent(MidiNoteOn, OnMidiNoteOn);
RegisterEvent(MidiNoteOff, OnMidiNoteOff);
```

We also have to fetch the fixture type ID of our Deskchannel fixture to be able to access it.

```
int nTypeId;
nTypeId = GetFixtureTypeId("Dskch");
if (nTypeId < 0)
{
            alert("Fixture Type does not exist! Aborting...\n");
            exit;
}
```

The most important part is calling the Suspend command to let the macro lay itself to sleep.

```
Suspend();
```

Finally, the functions that are to be called must be written. Both functions have to receive 3 int parameters. We're keeping it simple here, so only one Desk Channel fixture will be altered.

```
function OnMidiNoteOn(int p0, int p1, int p2)
{
            proClear();
            proSelectSingle(nTypeId, 1);
            SetPosition(65535);
            proLoadValue(0);
}

function OnMidiNoteOff(int p0, int p1, int p2)
{
            proClear();
            proSelectSingle(nTypeId, 1);
            SetPosition(0);
            proLoadValue(0);
}
```

The complete macro therefore looks as follows:

```
RegisterEvent(MidiNoteOn, OnMidiNoteOn);
RegisterEvent(MidiNoteOff, OnMidiNoteOff);

int nTypeId;
nTypeId = GetFixtureTypeId("Dskch");
if (nTypeId < 0)
{
            alert("Fixture Type does not exist! Aborting...\n");
            exit;
}
Suspend();

function OnMidiNoteOn(int p0, int p1, int p2)
{
            proClear();
            proSelectSingle(nTypeId, 1);
```

```
            SetPosition(65535);
            proLoadValue(0);
}

function OnMidiNoteOff(int p0, int p1, int p2)
{
            proClear();
            proSelectSingle(nTypeId, 1);
            SetPosition(0);
            proLoadValue(0);
}
```

Now all that has to be done is to start the macro.

You could of course make this example a lot more complex, like processing the function parameters that are passed when a function is executed. In this example you could switch on different fixtures dependent on which note has been played.

7 Device access

7.1 Starting and stopping cuelists

Cuelists can be started and stopped with the two e:script commands `CuelistStart` and `CuelistStop`. The only thing to keep in mind is that those two commands expect a zero-based parameter, so `CuelistStart(0)` will start cuelist number 1, `CuelistStart(1)` will start cuelist number 2 etc. The following short macro starts the first 5 cuelists:

```
CuelistStart(0);
CuelistStart(1);
CuelistStart(2);
```

Of course, we would handle that more elegantly with a for loop, as shown below:

```
int j;
for (j = 0; j <= 2; j++) CuelistStart(j);
```

If we want to stop all cuelists, we do not need to go to the trouble of writing a macro such as -

```
int i;
for (i = 0; i < 1000; ++i)
            if (CueGetCount(i) > 0) CuelistStop(i);
```

Instead, we can just use `CuelistStopAll` and this command will stop all cuelists, delayed by the release times, if existent. Alternatively, `ResetAll` will stop all cuelists at once with all release times set to 0.

7.2 Retrieving information about cuelists

The e:script language also allows to find out information about what is currently happening with cuelists. First of all, `CuelistGetName` simply returns the name of the particular cuelist. The cuelist number is passed as a parameter to the command. Note the parameter is zero-based again - `CuelistGetName(0);` will return the name of cuelist number 1. The statement `printf(CuelistGetName(1));` will output the name of cuelist number 2 to the logbook.

The command `CueGetCurrent` returns the number of the cue which is currently running (also zero-based - so if that function returns the value 33, it is actually cue 34 which is running at the time), or –1 if the cuelist isn't running at the time. For more advanced purposes we have `CuelistGetStatus`:

> CuelistGetStatus
> string CuelistGetStatus (int cuelist_id,int par);
>
> 'id' is the zero based number of the cuelist.
> The parameter 'par' selects the status field.
> 0: Play Status
> 1: Extended List Name
> 2: Submaster Position
> 3: Short List Name

The below macro checks the first 100 cuelists and prints the status to the logbook:

```
int i,j;
string myString;
```

```
for (i = 0;  i < 100; i++)
{
            myString = "";
            if (CueGetCount(i) > 0)
            {
                        myString += format("Cuelist %d \n", i +
1);
                        for (j = 0; j < 4; j++)
                        {
                         myString += format("    Par %d: ", j);
                         myString += CuelistGetStatus(i, j);
                         myString += format("\n");
                        }
                        myString += format("\n");
            }
            else
                        myString += format("Cuelist %d does not
exist\n",
                                                                  i + 1);

            printf(myString);
            printf("\n");
}
```

The first for-loop runs through the first 100 cuelists. It checks for every one of them if the cuelist actually exists by invoking `CuelistGetHandle` (since we wouldn't want to run `CuelistGetStatus` on a cuelist that doesn't exist - that could give us an error message and terminate the macro). About `CuelistGetHandle` - all we need to know for the moment is that it returns –1 if the cuelist doesn't exist, so we can check if the cuelist does exist with the statement `if(CuelistGetHandle(i) != -1) {…}` and everything inside the braces `{…}` will only be carried out if the cuelist exists. Note also the `else {…}` statement below these braces, which prints a message that the cuelist does not exist. This might be taken out of the macro using the comment symbols `//` because it might be very bothering to receive text for every cuelist that is not even running.

Now, inside the braces there is a second for-loop. This one runs `GetCuelistStatus` four times on the same cuelist, with a different value for the second parameter. All the information is added (nicely formatted) to a string variable `myString`. As we can see, the operator += also works for string variables.

```
myString = "Hello, ";
myString += "World";
myString += "\n";
```

will lead to exactly the same result as `myString = "Hello, World\n"`. At the end of the `Cuelist-GetStatus` macro, the variable `myString` is put out to the logbook using `printf` and another line break is added. Then the loop starts over again, `myString` is cleared and filled with more information, etc. The output of the macro in the logbook of the e:cue programmer would be similar to this:

```
Cuelist 1 does not exist
```

```
Cuelist 2
            Par 0: <new cue> [K]
            Par 1: QL2   background (1/1)
            Par 2: Full
            Par 3: QL2   background
Cuelist 1 does not exist
Cuelist 4
            Par 0: 0 [K]
            Par 1: QL4    (1/3)
            Par 2: Full
            Par 3: QL4
(etc. ...)
GetCuelistStatus done in 0.020 sec
```

7.3 Displaying cuelist information

The following macro fragment uses `CuelistGetStatus` to display information on an e:com terminal. For example, a user could press buttons on an e:com and the e:cue programmer would switch to a different cuelist, showing the name of that cuelist in the display of the e:com. There were two macros running in the Automation window: The first one was triggered by e:com events, responding to a key press on the e:com. The other one was triggered every second to refresh the e:com display, and is shown below:

```
string myString;
if (undefined(_currentCuelist))
{
            int _currentCuelist;
            _currentCuelist = 0;
}

myString = "-e:cue e:com Terminal-\n";
myString += CuelistGetStatus(_currentCuelist, 3);
myString += CuelistGetName(_currentCuelist,
                                        CueGetCurrent(_cur-
rentCuelist));
myString += "\nPress Button to Change.\n");
TerminalSend(0, 0, ""); // clear screen on all terminals
TerminalSend(0, 1, myString); // send message to all terminals
```

As you can see, this macro is doing basically the same thing as the one before, with the difference that the information isn't put into the logbook, but sent to an e:com terminal. Apart from that, note there is a global variable, `_currentCuelist`, which is set up and refreshed by the part of the macro which is not shown here. This part will be dealt with in a later chapter which deals with reading information from remote terminals. Note also that the e:com Terminal can display four lines - this is why one of the lines is removed with `//` to make room for another line which shows the name of the current cue.

7.4 Deleting cuelists

The command

```
CuelistDeleteAllCues(int cuelist_id);
```

will delete all cues from the cuelist specified by `cuelist_id` (which is zero-based).

7.5 Recording Cues

The command `RecordCue` does almost exactly the same as pressing the record button in the e:cue Programmer. It takes two parameters: First, the number of the cuelist which you want to record the cue into. This has to be a zero-based number. Second, it takes a string parameter for the name of the cue to be recorded.

Now it's very useful to be able to record a cue automatically, but the question is, how can we use e:script to put stuff into the programmer view in the first place? That is the question we want to deal with in the next chapter.

7.6 Accessing the Programmer View

In this chapter we will only talk about fixtures that have already been patched (we will deal with patching new fixtures in a later chapter). The approach taken by e:script is to mimic the user's behaviour , and follows this order:

- Fixtures are selected with `proSelectSingle` or `proSelectRange`.
- A value between 0 (0%) and 65535 (100%) is stored temporarily with `SetPosition`.
- Third, that value is loaded into one of the channels of the fixture with `proLoadValue`.
- Finally, the result can be stored inside a cuelist with `RecordCue` but for some purposes such as games etc. where e:script has complete control, this isn't necessary – after all, the programmer view has direct output.

(There is another way of directly accessing desk channels with the `LoadValue` function, but this is more complicated and will be dealt with later on.)

The following very simple example takes all RGBFader elements that are found in a show and sets them to white (i.e. 100% on all three channels – Red, Green and Blue). To try it, simply patch any number of RGBFader elements (from the Generic class in the library) and run this macro:

```
int fix_type, fix_count, i;

fix_type = GetFixtureTypeId("RGBFader");
if (fix_type < 0)
{
            MessageError("Fixture Type RGBFader not found in
show.");
            exit;
}

fix_count = GetFixtureTypeCount(fix_type);
printf("This many fixtures in show: %d \n", fix_count);

for (i=0; i <= fix_count; i++)
{
            proSelectOut();
            proSelectSingle(fix_type,i);
            SetPosition(65535);
            proLoadValue(0);
```

```
            proLoadValue(1);
            proLoadValue(2);
}
```

After declaring a few variables, the variable the fixture type ID of our “RGBFader”s is assigned to `fix_type` by using `GetFixtureTypeId`. If `GetFixtureTypeId` could not find any “RGBFader”s it will return a value below zero, which is how we can check for errors - we can present an error message to the user and exit.

Next, the variable `fix_count` is assigned the number of “RGBFader”s that are found in the show by using `GetFixtureTypeCount`. This requires the fixture type as parameter so we pass `fix_type` to it, and leave a little message in the logbook as to how many fixtures were found. Lastly, a for-loop repeats the same sequence for all the fixtures that were found. First of all, everything is deselected with `proSelectOut` – so if the user had any selections before running the macro it does not lead to any unexpected results. Then, `proSelectSingle` receives the parameters for the type and the ID number of the fixture to select. We are passing `fix_type` and the variable `i` from the for-loop to it – this way, we can cycle through all the fixtures.

Now the programmer knows which fixture we are accessing, so we can set a value that we would like to load using `SetPosition`. Remember, the range is 0 to 65535 (100%). And finally, we use `proLoadValue` three times, thus setting 65535 into the channels 1, 2 and 3. The parameter for `proLoadValue` is also zero-based - which means if we want to access the first channel, we write `proLoadValue(0)`. Note that in this example, we could have saved ourselves the for-loop by using `proSelectRange` instead of `proSelectSingle`:

```
proSelectOut();
proSelectRange(fix_type, 0, fix_count);
SetPosition(65535);
proLoadValue(0);
proLoadValue(1);
proLoadValue(2);
```

As you can see, `proSelectRange` takes three parameters – the fixture type and the first and the last fixture to select.

7.7 Setting Live FX

We use the command `SetFx` to temporarily store FX settings. Just as with `SetPosition`, the FX settings are loaded into a channel by using `proLoadValue`. This means you can use `SetPosition` and `SetFx` together, and afterwards load both the value and the FX into a channel with a `proLoadValue` statement.

```
SetFx
int SetFx(int func_id,int size,int speed,int offset,int ratio);

Set LiveFX Values
func_id = number of required function (0 = off)
size 100% = 4096,
        range [-5*4096 ... 5*4096]
        = LiveFX Window: [-500% ... 500%]
speed 1Hz = 1000, range [-20000 ... 20000]
        = LiveFX Window: [-20 ... 20]
offset 100% = 1000, range [-10000... 10000]
        = LiveFX Window: [-1000% ... 1000%]
ratio 100% = 4096, range [0 ... 4096]
        = LiveFX Window: [0 ... 100]
```

The first parameter, `func_id`, denotes the FX function you would like to set. The value is not a zero-based number. The following table shows function names.

1	Sin	8	Ping Pong
2	Cos	9	Halloween
3	Ramp	10 ... 13	Half Ramp 1 ... 4
4	Rectangle	14 ... 29	EQ bands 1 ... 16
5	Triangle	30 ... 34	EQ band X1 ... X4
6	Peek Up	35 ...	Video/screen capture channels, not used
7	Peek Down		

The parameter `size` takes values from –20480 to 20480 (-500% to 500%). The `speed` parameter takes values from -20000 to 20000. The following table helps estimating the values you can set:

speed	Hertz	Passes per second
1000	1 Hz	1
10000	10 Hz	10
100	0.1 Hz	10 s per pass

The parameter `offset` sets a phase shift off the effect. If you have two times the same Live FX function with different values for `offset`, they will oscillate out of sync. The `ratio` parameter “deforms” the mathematical function, changing the scaling of the left part relative to the scaling of the right part of the curve. For example, if you set a Rectangle function and the value 2048 for `ratio`, there will be a balanced relation between “on” and “off”. The minimum and maximum values are 0 and 4096 respectively.

7.8 Clear Fx

With `SetFx`, remember that before starting on another channel, you might want to erase the temporary value using `ClearFx`. Unless you do `ClearFx`, any channel that has a value assigned with `proLoadValue` also receives the effects. The example below sets an effect and position value to channel 1, then sets 0% and no effects to the two other channels:

```
SetPosition(32767);
SetFx(1, fx_size, -fx_speed, r, fx_ratio);
            //f,size,speed,offset,ratio
proLoadValue(0);
ClearFx();
SetPosition(0);
proLoadValue(1);
proLoadValue(2);
```

7.9 Example: making your own effect macro

Now that we got the tools to configure effects on fixtures using the e:script macro language, we want to use this to make our first effect creating macro. In the following example, we want to create an effect where a wave starts in the middle of an LED matrix, running to the outside, forming an increasing circle. We need to have some RGBfader patched, for now we assume that we have set up a 32x32 LED matrix. If you want to use the macro on a system with other dimensions, just change the `width` and `height` variables to appropriate values. This example will be more complex than the most we have done so far, so we build it up from the scratch. First, we declare a big bunch of variables and assign values to them:

```
int width, height, i, k, n;
int foffset, dx, dy;
int dropx, dropy;
int fix_type, fspeed, fratio, screen_id;
int brightness;
string fix_name, info;

width = 32;
height = 32;
fspeed = -400;
fratio = 4096;
brightness = 8192;
dropx = width/2;
dropy = height/2;
```

The variables `width` and `height` define the size of our matrix, so if you want your effect to be on a matrix with a different size, simply adjust the values. `i` and `k` will be used as control variables for the loops we will be using later on, while n will hold the value of the pixel currently addressed. `fspeed` and `fratio` will hold the values for speed and ratio we will pass to the `SetFx` command, with appropriate values assigned. `dropx` and `dropy` will hold the position where we want our wave to start, in this case the middle of our matrix. Finally, brightness holds our desired value for the color brightness we want. Now let us move on in the code:

```
screen_id = 1;
fix_name = "RGBFader";

fix_type = GetFixtureTypeId(fix_name);
                        //verify the fixture type exists.
if (fix_type<0)
{
            info=format("Fixture Type '%s' does not exist.",fix_
name);
            MessageError(info);
```

```
            exit;
}
```

In this part of the code, we make some formal preparations: The variable `screen_id` defines at which fixture we want to start. This is important if we have more than our LED matrix patched so that our matrix's addressing does not start at screen ID one. Here, we assume that our matrix starts at the first position, so we set `screen_id` to 1.

We need the type id of the fixture type "RGBFader" because we need it later on when we use `proSelectSingle` to select a single fixture. We can get it with the command `GetFixtureTypeId`. If we receive an id below zero, an error has occurred and we have to exit our macro. Now follows the most important part: We want a wave running from the middle to the outside in a circle. But how can we accomplish this? It is not as hard as you might think; the trick is to use a little math. Using two for-loops, we will go through the matrix, selecting a single fixture every time. With

```
dx = (i - dropy) * 100;
dy = (k - dropx) * 100;
```

we calculate the difference between the actual position and the middle of the matrix for both the horizontal axis and the vertical axis. We scale this difference by multiplying it with 100 (this has to be done so that the results differ a lot for different positions). With the command `radius` we can calculate the straight distance of our `dx` and `dy` to the centre of our coordinate system (the middle of our matrix). Every fixture in our matrix will receive the same Live FX with the same values, with only one difference: The offset will be our calculated distance from the middle of the matrix. A point further away from the centre will receive a bigger offset than a point near to the centre, which results in a more delayed display depending on the position. This does exactly what we want to have. Let us look at the code:

```
proClear();
for (k = 0; k < width; ++k)
{
            for (i = 0; i < height; ++i)
            {
                        n = k * height + i + screen_id;
                        proSelectOut();
                        proSelectSingle(fix_type, n);
                        dx = (i - dropy) * 100;
                        dy = (k - dropx) * 100;
                        foffset = radius(dx, dy);
                        SetPosition(brightness);
                        SetFx(9, 4096, fspeed, foffset, fratio);
                        proLoadValue(0);
            }
}
```

We first clear the Programmer view with `proClear` to make sure that our effect is the only thing to be displayed. Then we calculate our current fixture position depending on our control variables and add `screen_id` to it.

We use the Live Fx "halloween" which will generate a nice waveform. Remember that we assigned a negative value to `fspeed` so that the wave will float from the middle to the outside and not vice versa. Color value and effect will only be loaded to channel zero of each fixture because we want the wave to have a nice deep blue color. Now here is the complete macro written down in one piece:

```
int width, height, i, k, n;
int foffset, dx, dy;
int dropx, dropy;
int fix_type, fspeed, fratio, screen_id;
int brightness;
string fix_name, info;
width = 32;
height = 32;
fspeed = -400;
fratio = 4096;
brightness = 8192;
dropx = width/2;
dropy = height/2;
screen_id = 1;
fix_name = "RGBFader";
fix_type = GetFixtureTypeId(fix_name);
                        //verify the fixture type exists.
if (fix_type<0)
{
            info=format("Fixture Type '%s' does not exist.",fix_
name);
            MessageError(info);
            exit;
}
proClear();

for (k = 0; k < width; ++k)
{
            for (i = 0; i < height; ++i)
            {
                        n = k * height + i + screen_id;
                        proSelectOut();
                        proSelectSingle(fix_type, n);
                        dx = (i - dropy) * 100;
                        dy = (k - dropx) * 100;
                        foffset = radius(dx, dy);
                        SetPosition(brightness);
                        SetFx(9, 4096, fspeed, foffset, fratio);
                        proLoadValue(0);
            }
}
```

Feel free to experiment with the values! Try to use other effects, adjust the parameters and use different colors ore use another centre of effect. This will create very nice looking effects in a simple way. Instead of using the `radius` command, you can also try the command atan2. This will calculate the angle a point has relative to a centre position. This way, you can create beacon-like effects with a rotation going on.

7.10 Using LoadValue

The command `LoadValue` works similarly to `proLoadValue`. However, instead of putting an item

into the programmer view, `LoadValue` directly accesses the output of the e:cue programmer, as a cuelist would. Just like with `proLoadValue` though, first a value needs to be stored using `SetFx` or `SetPosition`. `LoadValue` requires the following parameters:

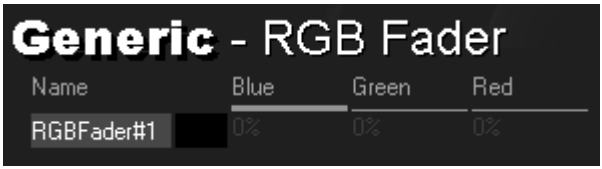
```
int LoadValue(int handle,int desk_id,
                                int fade_time,int delay_time,int htp);
```

Since `LoadValue` is very close to the core functions of the e:cue programmer, and since it is exercising the same access rights like a cuelist, `LoadValue` needs a currently running cuelist to get a handle from. We can either make and start our own cuelist with e:script or use one that is already there, and use `GetCuelistHandle` to retrieve a handle. `GetCuelistHandle` returns –1 if the cuelist isn't running, so we better make sure it does.

Second, `LoadValue` needs a to access "desk channel". The desk channel of any fixture can be retrieved with `GetDeskChannel`. For example, the following statement retrieves the base channel of RGB Fader number eleven:

```
GetDeskChannel("RGBFader",11);
```

In the case of the RGBFader, which has three channels, we need to add either 0, 1, or 2 to the result to get the correct desk channels for Blue, Green, or Red. Note that the channels are exactly as they are displayed when the fixture is put into the programmer view:



The parameters for `fade_time` and `delay_time` are values in milliseconds. If you set a number bigger than zero for `fade_time`, there will be a fade-in from the current status to the value you were passing. `delay_time` will wait for the specified time before the command is executed. Finally, the `htp` parameter specifies whether you want the command to be executed in the HTP (Highest Takes Priority) or LTP (Latest Takes Priority). Set `htp` to 1 to use HTP, set it to 0 to use LTP.

With HTP enabled, the highest value received gets priority control. With LTP, the last value that has been sent gets priority control. HTP cannot work if the channel has been explicitly defined as LTP in the Library Editor. In most cases, setting HTP to 1 will yield the highest possibility of the macro to behave like you would expect because LTP can produce some kind of "random" behavior".

Practical use of the `LoadValue` can be made whenever you want to create an effect by yourself and want it to fade in smoothly. This is not possible with proLoadValue while `LoadValue` provides this possibility easily.

7.11 Accessing cues

Creating and deleting cues

We have already seen that it is possible to record a cue with `RecordCue`. Apart from that, it is also possible to delete cues using `DeleteCue`. This function requires two parameters, the cuelist id

and the cue id – both zero-based. The statement `DeleteCue(0,4)` will delete cue number 5 from cuelist number 1.

Reading and setting cue properties

For retrieving information from cues, we use `CueGetProperty`. This requires three parameters: The cuelist id and the cue id (both zero-based) and the property that you want to read out. For setting the parameters for cues we use `CueSetProperty` which requires one additional parameter – the value that you want to set.

```
CueGetProperty and CueSetProperty
int CueGetProperty(int cuelist_id,int cue_id,
        string property);
int CueSetProperty(int cuelist_id,int cue_id,
        string property,int value);

property='Command':
        0 = MANUAL
        1 = Wait (Para0=Runtime[ms])
        2 = Timecode (Para0=Timecode[ms])
        3 = Until end of Cuelist (Para0=Cuelist id)

property='Para0':
        Set a parameter for the current command

property='InFadeTime':
        Fade in time [ms]
```

Please note that there are many more possible values for the `property` parameter, since you can access all cue settings. The complete list can be found in the e:script Reference. But for the moment, we are only interested in `Command`, `Para0` and `InFadeTime`. Let us assume for a moment you a programming a simple show with a few color changes that is supposed to run stand-alone. Because it is standalone, you do not want to have any of the timings set to MANUAL since that would stop your show. The following macro checks all cues. It is split into several parts with some explanations inbetween:

```
int i,k,count,manual_found,myProperty;
string message,list;

for (i = 0; i<255; i++)
{
            count = CueGetCount(i);
            if (count > 0)
            {
                        for (k = 0; k < count; k++)
                        {
                        if(CueGetProperty(i,k,"Command") == 0)
                        {
                                manual_found = 1;
```

```
                                        list += format("- Cuelist %d
                                                (%s), Cue %d (%s)\n"),
                                                i + 11,
                                                GetCuelistName(i), k+1,
                                                CueGetName(i,k));
                                }
                                }
                }
}

// SECOND PART OF THE MACRO

if (manual_found)
{
                message="Manual timing found in the following

cuelists:\n\n";
                message+=list;
                MessageWarning(message);
}
else
                MessageOk("Show OK!\nNo manual timings found.");
```

As the next step, we could offer the user to automatically replace any manual timings by WAIT timings. Below is only the second part of the macro.

```
// THIRD PART OF THE MACRO
if (manual_found)
{
                message = "Manual timing found in the following
cuelists:\n\n";
                message += list;
                message += "\nWould you like to set them to WAIT=1
second?";
                if(MessageYesNo(message))
                {
                                for (i=0;   i<255;   i++)
                                {
                                 count = CueGetCount(i);
                                 if (count>0)
                                 {
                                                for (k=0;  k<count;  k++)
                                                {
                                                                if(CueGetProperty(i,k,"C
ommand")==0)
                                                                {

CueSetProperty(i,k,
                                                                                "Com-
mand",1);
```

```
CueSetProperty(i,k,

"Para0",1000);
                                                    }
                                                }
                                            }
                                            }
                                }
                                else exit;
}
else MessageOk("Show OK!\nNo manual timings found.");
```

For simplicity's sake, we just set the WAIT timing to one second - we could have also offered the user a dialogue and asked him for the WAIT timing he would have preferred. Note how there are two `SetCueProperty` statements. The first one passes "Command" as the property and sets the timing to WAIT by passing 1 as the value (see function description above). The second one passes "Para0" as the property – a variable parameter depending on the status of "Command". In this case the value affects the timing, 1000 ms = 1 second. A few more examples of how to use `Command` and `Para0`:

```
CueSetProperty(0,0,"Command",2);
CueSetProperty(0,0,"Para0",1000);
```

The above sets the timing of the first cue of the first cuelist to SMPTE timecode 0:00:01.00.

```
CueSetProperty(0,0,"Command",3);
CueSetProperty(0,0,"Para0",1);
```

The above sets the timing of the first cue to "Until the end of Cuelist 2".

7.12 Example: generate single colors macro

The following example uses what we have learned so far to generate a color-changing cuelist. Note how, for simplicity, we have set a lot of variables manually; we could also have added some user interaction to ask the user for the variables they like.

```
int i, k, n, count;
int width, height;
int fix_type, target_list_id;
int fade_in_time, wait_time;
string info;
// Some parameters are set manually here for simplicity,
// but we could also get them off the user:
width = 32;
height = 8;
fix_type = GetFixtureTypeId("RGBFader");
target_list_id = 0;
fade_in_time = 1000;
wait_time = 3000;
info = format("This macro will generate colours.\nCuelist %d
                          will be deleted. Proceed?", target_list_
id+1);
if(!MessageYesNo(info)) exit;
CuelistDeleteAllCues(target_list_id);
ClearFx();
```

```
proClear();
for (k = 0; k < width; ++k)
{
            for (i = 0;  i < height; ++i)
            {
                        n = k * height + i + 1;
                        proSelectOut();
                        proSelectSingle(fix_type,n);
                        SetPosition(0xffff);
                        proLoadValue(1);
            }
}
RecordCue(target_list_id,"Green");
proClear();
for (k = 0; k < width; ++k)
{
            for (i = 0; i < height; ++i)
            {
                        n = k * height + i + 1;
                        proSelectOut();
                        proSelectSingle(fix_type,n);
                        SetPosition(0);
                        proLoadValue(0);
                        SetPosition(0xffff);
                        proLoadValue(1);
                        proLoadValue(2);
            }
}
RecordCue(target_list_id, "Yellow");
proClear();
for (k = 0;  k < width; ++k)
{
            for (i = 0;  i < height; ++i)
            {
                        n = k * height + i + 1;
                        proSelectOut();
                        proSelectSingle(fix_type, n);
                        SetPosition(0);
                        proLoadValue(0);
                        proLoadValue(1);
                        SetPosition(0xffff);
                        proLoadValue(2);
            }
}

RecordCue(target_list_id, "Red");
proClear();

for (k = 0; k < width; ++k)
{
```

```
            for (i = 0; i < height; ++i)
            {
                        n = k * height + i + 1;
                        proSelectOut();
                        proSelectSingle(fix_type, n);
                        SetPosition(0);
                        proLoadValue(1);
                        SetPosition(0xffff);
                        proLoadValue(0);
                        proLoadValue(2);
            }
}
RecordCue(target_list_id, "Magenta");
proClear();
for (k = 0;  k < width; ++k)
{
            for (i = 0;  i < height; ++i)
            {
                        n = k * height + i + 1;
                        proSelectOut();
                        proSelectSingle(fix_type, n);
                        SetPosition(0);
                        proLoadValue(2);
                        SetPosition(0xffff);
                        proLoadValue(0);
                        proLoadValue(1);
            }
}
RecordCue(target_list_id, "Cyan");
proClear();
count = CueGetCount(target_list_id);
if (count > 0)
{
            for (i = 0; i < count; ++i)
            {
                        CueSetProperty(target_list_id, i,
                         "InFadeTime", fade_in_time);
                        CueSetProperty(target_list_id, i, "Com-
mand", 1);
                        CueSetProperty(target_list_id, i,
"Para0", wait_time);
            }
}
```

8 Processing input and output

8.1 Serial I/O

In chapter 9.3.3, within the example about string intersection, we processed a simulated serial input string. In this chapter, we want to pick up this topic and explain how to receive real input, as we already know how to analyze the received input.

To receive a serial input that arrived via RS-232, we make use of the command `GetSerialString`. Let us take a look at the definition:

> GetSerialString
> string GetSerialString(int handle, int-ref length);
>
> handle – driver handle
> Length - reference parameter and will be set to the length of the received string or -1 if input buffer was empty.

The most important thing to know is what the driver handle is and how to retrieve it: In the e:cue Programmer, every device you add to a show will get a unique handle which acts as an identifier for that device. To retrieve it, you need the alias name of the particular device. The command `DriverGetHandle` needs this name as its single parameter and returns the handle related to the device name.

The return value of this command is obviously the received serial input. The parameter `length` works like an additional return value: `GetSerialstring` will write the length of the received string into the parameter length. This will become very important as we need to detect when an input signal has been completely received (this will be explained below).

But how do we recognize when a serial input waits for us to be caught? This is easy as we can build up a trigger rule that reacts on incoming data on a given serial port You can find more about triggers inside the e:cue Programmer system manual. The trigger will call our macro that will then receive and process the input as we want to.

A serial input can consist of multiple lines, separated with the control character '\r' (carriage return). `GetSerialString` will only return one line each time it is called. To receive the complete message, we must call the command repeatedly until the length parameter becomes -1. As it is not possible to create a string array, we have to process the input data line by line inside the loop (e.g. convert it into an int if possible to store the data into an int array). The following code shows a general structure of how to receive a multiline input:

```
// This macro has to be triggered on serial input
int length;
string input;
int handle;
int i;

handle = DriverGetHandle("serial#1");
i = 0;
```

```
while (i != -1)
{
            input = GetSerialString(handle, i);
            […] // further process the received input line
}
```

We did already show how to process a string in chapter 9.3. As the input depends on the device that sends the data and as you have to know what you want to do with it, you have to work out your own way of processing the input.

You can also send data to a serial port with `SendString`. The syntax is as follows:

```
int SendString(int id,string message);
```

The parameter id specifies the port number you want to send the data to. The parameter message contains the data itself. `SendString` will return 1 if the sending succeeds, 0 if an error happens. You see, this is simple. The only complicated thing about this is how the message has to look like. This is because it depends on the device you want to send the data to. Often, this will look very cryptic and it also might be very complicated to build up the string (you might have to use `setchar` to set up the string one char after another).

8.2 MIDI

As well as sending and receiving serial input via RS-232, it is also possible to send MIDI data. As we do not want to digress from our main topic, which still is macro programming, we suppose that you know the basics about MIDI and what can be sent with the MIDI protocol.

Standard MIDI messages

There are several different ways to send data to a certain MIDI port, mostly depending on what kind of data to send. The simplest message is a simple note on message and can be accomplished with the command `MidiNoteOn`. Look at the syntax in the following table:

MidiNoteOn int MidiNoteOn(int midi_handle,int channel,int note,int velocity); midi_handle - Handle number for the midi output device. channel - Zero based midi channel, range [0..15] note - Note number, range [0..127] velocity - Velocity value, range [0..127]

The parameter `midi_handle` specifies the device to which you want to send the data. This concept is exactly the same as the one you have met in the section about serial I/O: To retrieve the handle, again we need the command `DriverGetHandle`. This needs the device alias name as a parameter. The parameters channel, note and velocity are self-explaining as long as you are familiar with the MIDI standard: They specify the note, which MIDI channel and what velocity to use. The return values of the `MidiNoteOn` command are 1 for success and 0 otherwise.

To turn a certain note off, use `MidiNoteOff`. This works equivalent with the same syntax, but will obviously turn off the specified note.

The MIDI protocol allows program and control change messages. Both can be done with e:script: the particular commands are `MidiSendProgChange` and `MidiSendCtrlChange`. Let us first reckon `MidiSendProgChange`:

MidiSendProgChange
int MidiSendProgChange(int midi_handle,int channel,int val);

midi_handle - Handle number for the midi output device.
channel - Zero based midi channel, range [0..15]
val - Program number, range [0..127]

Again, we need the handle to address a certain MIDI device. The parameter `channel` specifies the MIDI channel while `val` indicates the desired MIDI program number.

`MidiSendCtrlChange` works quite similar but has a little differing parameters. Let us have a look at it:

MidiSendCtrlChange
int MidiSendCtrlChange(int midi_handle,int channel,int low, int high);

midi_handle - Handle number for the midi output device.
channel - Zero based midi channel, range [0..15]
low - Controller number, range [0..127]
high - Controller level value, range [0..127]

The parameters `midi_handle` and `channel` are the same like in `MidiSendProgChange`. The parameters `low` and `high` specify the desired controller number and controller level. Both commands will return a 1 in case of a success and a 0 if any error occurs.

Advanced MIDI messages

To achieve a greater flexibility in MIDI messaging like hardware-dependant system messages, e:script offers two general-purpose MIDI message commands, `MidiSend` and `MidiSendBinary`. `MidiSend` dispatches a single unsigned byte to a given port, with the simple syntax

```
int MidiSend(int midi_handle, int byte);
```

where `midi_handle` specifies the MIDI device and `byte` contains the message itself.

`MidiSendBinary` sends a message of up to 255 characters to a MIDI device. The following table contains the syntax:

MidiSendBinary
int MidiSendBinary(int midi_handle,string message,int length);

midi_handle - Handle number for the midi output device.
message - Binary coded midi message
length - Message length, range [1..255]

The parameter `midi_handle` does act as usual. The string `message` holds the message you want to deliver and `length` specifies the message length. To set up the message string, just set the single bytes with `setuchar`. This is necessary especially when you want to use control characters because the string will not be interpreted like in `printf` but sent directly.

As this is a kind of general-purpose message, you will have to know what to send and set up the message syntax by yourself to make sure the receiver understands the message.

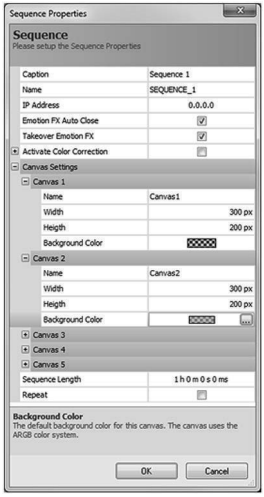
`MidiSend` and `MidiSendBinary` both return 1 in case of a successful sending and 0 in case of an error, just like the other MIDI related commands.

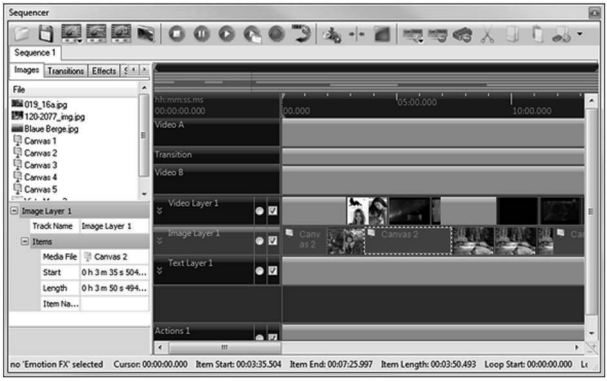
9 Canvas objects in the Sequencer

9.1 Background

With the Programmer version 5.5 SR1 the Emotion FX sequence is now able to handle canvas objects. Canvas objects are individual drawing media objects that can be manipulated with e:script commands. All available for canvas manipulation can be found in the e:script Language Reference in Sequencer Classes.

Canvas objects can be used like an any other image object. Place the item by a drag and drop operation on a Media Layer Track. The new Media Preview window will show the current state of the Canvas object. Each sequence can handle five independent Canvas objects. All of them work like image objects.





9.2 Usage

After connecting Emotion FX to the Programmer, the Emotion FX generates five initial Canvas objects. Now you can manipulate the Canvas object content by e:script:

```
// Canvas script
int hSeq = SequenceGetAt(9);
int hCanvas = SequenceCanvasGetAt(hSeq, 0);
if (hCanvas == 0)   return;  // Error

int iPenArgb = RGBA(200, 200, 200, 255);

SequenceCanvasBeginPaint(hCanvas);
SequenceCanvasClear(hCanvas, RGBA(0, 0, 200,100);
SequenceCanvasDrawLine(hCanvas, iPenArgb, 2, 50, 50, 200, 200);
SequenceCanvasEndPaint(hCanvas);
```

This executes:

- Select a Sequence object
- Select a Canvas object from the Sequence
- Start drawing commands with BeginPaint( )
- Execute drawing commands
- End drawing with EndPaint( ) and transfer changes to Emotion FX

The command SequenceCanvasEndPaint( ) sends the drawing commands between the Start/End directly to the connected Emotion FX. In case of neglecting SequenceCanvasClear, the previous drawing will be overdrawn. Please note that the Programmer sends only the drawing commands to the Emotion FX, never the drawn bitmap.

The Canvas object can be exclusively used for non-dynamic changes, like shade some image areas. It is not possible to create live effects with this. Use the Canvas objects with care, because these can stress the Emotion FX system. Use only small dimensions.

10 Reading macro arguments

We have already discussed the use of triggers to be able to react to certain events. Now we want to deepen this subject as we have the possibility of some advanced use of triggers: When a macro has been called by a trigger or by another macro, it can receive multiple arguments.

10.1 Calling a macro from another macro

It is possible to execute a macro from within another macro. When we do this, first the called macro will be executed and then the rest of the macro that has called the other one. They will not run simultaneously.

How do we call a macro from a running macro? Intuitively, we have a macro command `Call` for this. The syntax of this command is as follows:

```
int Call(string name, int arg1, int arg2, int arg3);
```

The parameter name holds the name of the macro to be called. It is very important do know that you can only call macros that are stored within the current show file. If you have more than one macro with the same name, the `call` command will choose the topmost macro inside the macro list. The three integer parameters arg1, arg2 and arg3 are optional. These are the parameters the called macro can receive (how to do this will be explained later on). There is a rub in it: As said before, these parameters are optional. However, you can only choose to use none or all of them. This is no big problem because if you want to pass just one or two parameters, you can fill up the rest with dummies.

Now that we know how to call a macro from another one and how to pass parameters to it, the thing left to know is how to make the called macro receive these parameters. This will be explained in the next section. It is possible to call another macro from an already called macro but that is also the limit; a recursion depth of 15 is the maximum. If you exceed this limit, the programmer will stop the macro execution and write an error into the logbook. To prevent this, always attend to this. Excessive cross-macro-calling will also make the code much harder to understand.

10.2 Receiving arguments in a called macro

To read the arguments passed to a macro we make use of the command `getarg`. It is very simple: to get the first argument the macro received, we say

```
a = getarg(0);
```

This will store the first argument into a given integer variable `a`. The second and third parameters can be stored by changing the parameter to 1 or 2 respectively.

Let us take a look at a very simple example: First, we write a short macro which will call another macro and pass three parameters. The called macro will receive the parameters and simply put them out via `printf`. Here we go:

```
// Call a macro named "callme" with 3 arguments passed
int a, b, c;

a = 1;
b = 2;
c = 3;
Call("callme", a, b, c);
```

Now let us view the macro that will be called from the macro above:

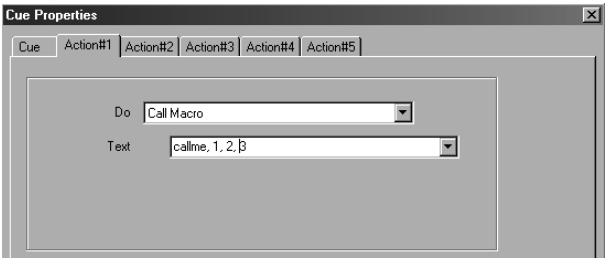
```
// alled from another. It is important to name it correctly.
int arg1, arg2, arg3;

arg1 = getarg(0);
arg2 = getarg(1);
arg3 = getarg(2);
printf("Received the following parameters: %d, %d, %d\n", arg1,
arg2, arg3);
```

When you start the first macro, it will call the second one, and the output in the logbook will be “Received the following parameters: 1, 2, 3” along with some information about the current recursion depth. As you can see, this works very simple and can be very practical.

10.3 Passing arguments on macro call

When you call a macro using a cuelist or triggers, you can pass parameters to it. To do so, simply attach the parameters to the macro name, separated with commas. Take a look at the following picture:



This example shows how to call a macro named `callme` with three parameters 1, 2 and 3. Just like in the example before, the called macro can retrieve these arguments with the command `getarg`. This way, you can pass up to 5 arguments to a macro you want to call.

10.4 Reading arguments from Triggers

A very special case is a macro being called by a trigger. In this case, arguments will be passed automatically, containing information about the certain trigger that initiated its execution. The following Table shows what information will be delivered on the various trigger events.

Trigger	getarg(0)	getarg(1)	getarg(2)	getarg(3)
Initialization	0	0 = Load Show 1 = Reset all	0	0
Periodic	1	Day of week (0 – 6; Sunday = 0)	Hours after midnight (0 – 23)	Minutes after hour (0 – 59)
Hourly	2	Hour of day (0 – 23)	Minutes after hour (0 – 59)	Seconds after minute (0 - 59)
Cuelist	3	Zero based cuelist id (0 for QL#1)	0=STOP 1=START 2=IDLE (Idle message are generated automatically)	0
Faderunit	4	Source Device 0= Fader Unit	0=Key released 1=Key Pressed	Key Number
Midi Note On	5	Zero based Midi channel (0 - 15)	Midi Note Number (0 - 127)	Velocity (0 – 127)
Timecode Event	6	0	0	0
Label	7	Zero based label number	0	0
Sunrise Timer	1	Day of week (0 – 6; Sunday=0)	Hours after midnight (0 – 23)	Minutes after hour (0 – 59)
Every Second	2	Hour of day (0 – 23)	Minutes after hour (0 – 59)	Seconds after minute (0 - 59)
Terminal Event	10	Terminal ID	1 = Init 2 = Key pressed 3 = Network Parameter Update	Key number for key press event (1 - 15), otherwise zero
Terminal Event ME2 Input Module (LCE)	10	Terminal ID	3 = Input Raise 4 = Input Fallback	**number of raise/ fallback input pin (1 - 16)**
Serial Port	11	Zero based port number 0 for Serial1 1 for Serial2	ASCII Code	Decoded Number Value
Midi Note Off	12	Zero based Midi channel (0 - 15)	Midi Note Number (0 - 127)	Velocity (0 - 127)
Midi Control Change	13	Zero based Midi channel (0 - 15)	Midi Controller Number (0 - 127)	New Controller Value (0 - 127)

Midi Program Change	14	Zero based Midi channel (0 - 15)	Midi Program Number (0 - 127)	0
Action Pad Event	15	Item ID	0 = Key Pressed 1 = Key Released 2 = Dragging	Cursor position within the Action Pad
Peer Event	16	Peer ID (1 – 255)	Event Class	Parameter (1 – 255)

The first argument will always deliver the trigger number with each trigger type having a unique identification number. The arguments two, three and four depend on the actual trigger that called the macro: The various time-based triggers for example deliver information about the time the trigger was activated. In contrast, Triggers activated by incoming data like serial input or MIDI events will give information about the received data. This makes it easy to react on the data transmitted via macro commands.

Example: You might want the Programmer to light up a certain fixture depending on a note played on a connected MIDI keyboard. To simplify this example, we assume that we do not care about the actual MIDI channel and the velocity value as well. We just have 128 fixtures patched, each to be lightened up when the appropriate note is played and darkened again when the note is released. We need two triggers to achieve this: One trigger has to react on any Midi Note On input while the other one has to react on a Midi Note Off. Both triggers are to call the same macro.

Inside the macro we will read out the delivered arguments and depending on which trigger called the macro, we will lighten or darken the appropriate fixture. We assume using RGBFader fixtures. Below you can see the code of the macro, followed by some explanation:

```
// This macro has to be called by Midi Note triggers
string fix_name, info;
int fix_type, screen_id;
screen_id = 1;
fix_name = "RGBFader";

fix_type = GetFixtureTypeId(fix_name);  //verify that the defined fix-
ture type exists
if (fix_type < 0)
{
            info = format("Fixture Type '%s' does not exist.",fix_
name);
            MessageError(info);
            exit;
}

int arg0, arg1;
arg0 = getarg(0);
arg1 = getarg(1);
proSelectOut();
```

```
switch(arg0)
{
            case 5:
                        proSelectSingle(fix_type, arg1 + screen_
id);
                        SetPosition(65535);
                        proLoadValue(0);
                        proLoadValue(1);
                        proLoadValue(2);
                        break;
            case 12:
                        proSelectSingle(fix_type, arg1 + screen_
id);
                        SetPosition(0);
                        proLoadValue(0);
                        proLoadValue(1);
                        proLoadValue(2);
                        break;
            default:
                        alert("Macro has to be called by a Midi
                                    Note trigger!\naborting…
\n");
                        exit;
}
```

We first declare the needed variables and check if the defined fixture type exists. If it does, we fetch the first two trigger arguments and save them inside `arg0` and `arg1`. All functionality resides inside the switch: we first check the value of `arg0` and react depending on its value. In case of a Midi Note On trigger (identified by a 5) we select the fixture appropriate to the note played (this value can be found in `arg1`) and lighten it up. If a Midi Note Off trigger (identified by a 12) called the macro the fixture appropriate to the turned off note will be darkened. If arg0 has neither value 5 nor 12 (that means the macro was not called by a Midi Note trigger) the macro will abort and place an error message into the logbook.

This is just a simple example of a practical use for reading arguments from triggers. With more effort, very complex behavior can be achieved.

11 Accessing the Action Pad

The Action Pad is a very comfortable and flexible control interface which can be set up without having to write macros. It is already very powerful without the use of e:script macros but in combination, especially when using triggers, a lot more is possible.

For this chapter, we imply that you know the basics of how to use the Action Pad, in detail how to add some elements to an Action Pad and add functionality.

11.1 Addressing single elements

Several e:script commands exist to manipulate single elements within the Action Pad. Before we go into detail, the first thing to know is how these items are addressed. First of all, if you add any item to the Action Pad, it automatically receives a member code. The codes are assigned continuously, starting with a zero for the first item. The numbers can only be seen in edit mode; you will notice them easily as they are near the top of each item.

The second important aspect is that the Action Pad may consist of several tabs. In each tab, the member codes reiterate. That means that you will have items with the same identification number on each tab, e.g. a single action button with ID 0 on tab one and an auto text display with the same ID on tab two.

To address a specific Action Pad element, you will first have to switch to the correct tab. This can be done in two ways: You can use `ActionPadSetCurrentPage` to select the desired tab via its page number. `ActionPadSetCurrentPage` has only one parameter, that is, needless to say, the page number itself. If you have named your tabs, it is possible to select a tab with `ActionPadSetCurrentPageByScriptID`. This command needs a string as a parameter, containing the name ID of the desired tab. It is not case sensitive and if more than one tab with the same name exists, the leftmost tab will be selected. When you are on the correct tab, you are ready to manipulate single items, as shown in the next section.

In some cases, it might be useful to get to know which is the active Action Pad tab. For this cause, you can use the command `ActionPadGetCurrentPageIndex`. This will return the identification number of the tab that is currently active.

11.2 Manipulating Action Pad items

The commands we will explain now all need the member code to address a certain item. How to retrieve the member code for an item has been explained above.

The most obvious manipulation done to an Action Pad item will be to change the information text each item holds. To do so, we use `ActionPadSetItemText`. In the following table, you can see the syntax of this command:

ActionPadSetItemText
int ActionPadSetItemText(int member_code,string text);

member_code - Action pad item member code (The small number on the top of the item, visible in edit mode only)
text – the new item text for the selected Action Pad item

The parameter `member_code` needs the appropriate ID to address a designated item. The parameter `text` will hold the new item text for the Action Pad item you want to change.

It is also possible to change the background colors of the Action Pad items and the text colors as well. Two colors exist for both: One is the primary color that will come into place in most instances. The secondary color takes place only in special circumstances, e.g. when a flash button is pressed it will use the secondary background and text colors. To change the colors, we have two commands available: `ActionPadSetItemBackgroundColors` and `apSetTextColors`.

In the following table, you can see the syntax of the command `ActionPadSetItemBackground-Colors`. As `ActionPadSetItemTextColors` has exactly the same syntax, there is no need for a table especially for that command:

`ActionPadSetItemBackgroundColors`
int `ActionPadSetItemBackgroundColors`(int member_code,int col_t0,int col_t1);

member_code - Action pad item member code (The small number on the top of the item, visible in edit mode only)
col_t0 - 24 bit text color (can be created by calling the function RGB)
col_t1 - 24 bit text color (can be created by calling the function RGB)

The parameter `member_code` has the same function as we already know from `ActionPad-SetItemText`. The parameters `col_t0` and `col_t1` will hold the color values for both the primary and the secondary color. You might have already noticed while looking at the table that both parameters contain a single 24-bit color value. This value has to be created by calling the command RGB that will build up such a value of three 8-bit values for the red, green and blue components a color has.

In combination with triggers, the ability to manipulate Action Pad items can be very useful. You can display context sensitive information, e.g. depending on some input the Programmer receives and at the same time dynamically highlight designated information. This can often make the work with the Action Pad much simpler.

11.3 Action Pad Autotext

Nearly all Action Pad items implement the Auto Text feature. Auto Text offers a dynamic interpretation of Action Pad item text to display situation-based content without having to change the information manually. Auto text is very powerful; you can display most of the status information the Programmer delivers. This is not an e:script feature, but as you might have expected, you can make use of Auto Text with `ActionPadSetItemText`. To give a start, the following table shows all possible information you can display:

category	displayed text	syntax	additional information
cuelist	current cue	<cuelist 1 current>	The cuelist number (here: 1) is not zero based.
	previous cue	<cuelist 1 prev>	
	next cue	<cuelist 1 next>	
	name	<cuelist 1 name>	
	flags	<cuelist 1 flags>	
	status	<cuelist 1 status>	
	submaster	<cuelist 1 submaster>	

fader	current cuelist number	<fader 1>	The fader number (here: 1) is not zero based.
	current cuelist name	<fader 1 name>	
	current cue	<fader 1 current>	
	next cue	<fader 1 next>	
	flags	<fader 1 flags>	
	status	<fader 1 status>	
	submaster	<fader 1 submaster>	
mutual exclude group	current cuelist number	<mutex 1>	The mutual exclude group number (here: 1) is not zero based.
	current cuelist name	<mutex 1 name>	
	current cue	<mutex 1 current>	
	next cue	<mutex 1 next>	
	flags	<mutex 1 flags>	
	status	<mutex 1 status>	
	submaster	<mutex 1 submaster>	
page name	current	<page current>	Displays the current, previous or next Cuelist Page.
	previous	<page prev>	
	next	<page next>	
grand master	name	<master 0 name>	Master 0 is always the Grand Master.
	status	<master 0 status>	
versatile master	name	<master 1 name>	Numbers from 1 to 63 address the different Versatile Masters.
	status	<master 1 status>	
time	current weekday	<nicetime %A>	These are the predefined menu items for this. All other nicetime format parameters are also possible.
	current time	<nicetime %X>	
	current date	<nicetime %x>	
network	network parameter	<network 1>	The first number specifies the network parameter (Not zero based). With bit and bool you can query a single bit of the particular network parameter (This works zero based).
	network parameter bit	<network 1.0>	

	network parameter bool	<network 1.0 “TRUE” “FALSE”>	“TRUE” and “FALSE” will show up instead of pure numbers. Both can be replaced with other strings (e.g. “On” “Off”).
media	media status	<media status 0>	This shows the status of the internal Media Player.
driver	name	<driver “aliasname” name>	The “aliasname” has to be the specified name for the particular device. When querying the logbook, the negative number specifies the line to be given out. A ‘-1’ will display the last line, while a ‘-2’ will display the line before, and so on.
	address	<driver “aliasname” address>	
	status	<driver “aliasname” status>	
	live property int	<driver “aliasname” int “property_name”>	
	live property string	<driver “aliasname” string “property_name”>	
	logbook	<driver “aliasname” logbook -1>	
logbook	main logbook	<logbook main -1>	The negative number works just like in the logbook query of the driver category.
	e:net logbook	<logbook network -1>	
special char	‘less than’	\<	Because the characters are used to indicate Auto Text, you need to use this syntax.
	‘greater than’	\>	

To display the requested information, all you have to do is put the particular Auto Text inside your string and assign it to the Action Pad item you want using `ActionPadSetItemText`.

As a very simple example, we want to let the first Action Pad item display information about the current cue, followed by the current cuelist status, both of cuelist 1.

```
ActionPadSetItemText(0,”Status of cuelist 1: <cuelist 1 status>\n
                Flags: <cuelist 1 flags>”);
```

As you can see, control characters like ‘\n’ and additional text is still possible; the Auto Text embeds itself perfectly.

Nicetime format string codes

The following table contains the information needed to build up a time information string. This relates to the auto text time commands.

%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Date and time representation appropriate for locale.
%d	Day of month as decimal number (01 ... 31).
%H	Hour in 24-hour format (00 ... 23).
%I	Hour in 12-hour format (01 ... 12).
%j	Day of year as decimal number (001 ... 366).
%m	Month as decimal number (01 ... 12).
%M	Minute as decimal number (00 ... 59).
%p	Current locale,s A.M./P.M. indicator for 12-hour clock.
%S	Second as decimal number (00 ... 59).
%U	Week of year as decimal number, with Sunday as first day of week (00 ... 53).
%w	Weekday as decimal number (0 .. 6; Sunday is 0).
%W	Week of year as decimal number, with Monday as first day of week (00 ... 53).
%x	Date representation for current locale.
%X	Time representation for current locale.
%y	Year without century, as decimal number (00 ... 99).
%Y	Year with century, as decimal number.
%z, %Z	Time-zone name or abbreviation; no characters if time zone is unknown.
%%	Percent sign.
$H0	Twilight_AM Hour (00 ... 23).
$M0	Twilight_AM Minute (00 ... 59).
$H1	Sunrise Hour (00 ... 23).
$M1	Sunrise Minute (00 ... 59).
$H2	Noon Hour (00 ... 23).
$M2	Noon Minute (00 ... 59).
$H3	Sunset Hour (00 ... 23).
$M3	Sunset Minute (00 ... 59).
$H4	Twilight_PM Hour (00 ... 23).
$M4	Twilight_PM Minute (00 ... 59).

11.4 New e:script commands since LAS 6.0

With LAS 6.0 a new set of Action Pad commands were introduced like ActionPadSetColorPicker-Color(), ActionPadSetColorPickerHue(), ActionPadSetColorPickerSaturation() etc.

See the e:script reference in the Programmer for details.

12 Terminals

12.1 Retrieving terminal input

The e:script language offers a connection between the e:cue Programmer and available e:com Terminals (including the e:node). You already got acquainted with reading arguments from triggers and this works just the same when dealing with terminals: Add an e:com Terminal Device in your device Manager, create a Trigger rule reacting on the desired Terminal input (most common would be a key message). Then use the command `getarg` to receive terminal ID, event class and key number. As an example, the following macro will retrieve the key numbers that have been pressed on the connected terminal and, if one of the keys F1 – F10 has been pressed, start or stop the associated cuelist (depending on its current status). The following is a great and simple example for remotely controlling the e:cue Programmer via e:com Terminal:

```
int key;

if (getarg(0) != 10)  //if it is no terminal event, abort
{
            printf("Wrong Trigger ID. Aborting...\n");
            exit;
}

if (getarg(2) != 2)  //if it is no key pressed event, abort
{
            printf("Wrong Event Class. Aborting...\n");
            exit;
}

key = getarg(3);
if (key >= 6 && key <= 15) // test if an F-Key has been pressed
{
            key -= 6; // adjust key number to cuelist number
            if (!strlen(CuelistGetStatus(key, 0)))
                        CuelistStart(key);
            else
                        CuelistStop(key);
}
```

At first, we verify that the macro has been called by the correct trigger and event class (a terminal event delivers getarg(0) = 10, a terminal key press event delivers getarg(2) = 2. For more details about trigger parameters, go back to the chapter about reading arguments from triggers, where you can find a complete list of all trigger parameters).

With `key = getarg(3);` we retrieve the key number and store it inside the variable `key`. As the key numbers of the F-keys go from 6 to 15, we must subtract 6 so that the key number is in line with the Cuelist numbers. Also the rest has no magic behind it: `CuelistGetStatus` will return an empty string if the cuelist is not running, so if it does, we start the appropriate cuelist. If the string is not empty, we stop it. That is it, with this short and simple macro we added a practical remote functionality to our Programmer.

12.2 Sending information to the terminal

Now that we know of the possibility to retrieve information from a connected e:com Terminal device, we can do nearly all we want with our terminal. But there is still a nice feature that might come in handy: The e:com Terminals have LCD-Displays attached; you can display status information there using e:script commands. This allows feedback for users operating on a terminal device.

The e:script command that we need to write on the display is `TerminalSend`. Its syntax is as follows:

```
int TerminalSend(int handle, int line_id,string message);
```

The handle should be already well known. We retrieve it using the command `GetDriverHandle`. The parameter `line_id` needs the display line we want to write our message to. To clear the display before writing the new message, set `line_id` to -1. The string `message` obviously contains the message itself. Now that we already have an existing macro for remote cuelist control, we want to extend it so that the connected terminal device will display Information about what happened. Here we go:

```
int key, handle;
string info;
handle = DriverGetHandle("term#1");
if (getarg(0) != 10)  //if it is no terminal event, abort
{
            printf("Wrong Trigger ID. Aborting...\n");
            exit;
}
if (getarg(2) != 2)  //if it is no key press event, abort
{
            printf("Wrong Event Class. Aborting...\n");
            exit;
}
key = getarg(3);
if (key >= 6 && key <= 15) // test if an F-Key has been pressed
{
            key -= 6; // adjust key       number to cuelist number
            if (!strlen(CuelistGetStatus(key, 0)))
            {
                        CuelistStart(key);
                        info = format("Started Cuelist #%d", key
+ 1);
                        TerminalSend(handle, -1, info)
            }
            else
            {
                        CuelistStop(key);
                        info = format("Stopped Cuelist #%d", key
+ 1);
                        TerminalSend(handle, -1, info)
            }
}
else
            TerminalSend(handle, -1, "");
```

Not much has changed: Whenever a cuelist starts or stops, we use TerminalSend to inform the

Terminal operator what just happened. If a key apart from the F-keys has been pressed, the display is cleared.

13 Saving variables to XML

When running in User and respectively in Kiosk Mode, an e:cue Programmer show is considered as persistent, non-changeable data. This is for safety reasons; in case that you want to grant users limited control over a lighting installation (in most cases via the Action Pad) but nothing more than that.

Nevertheless, there might be the need to save some status information even in User/Kiosk Mode: As an example, assume that you control the lighting of several rooms of a building using Action Pad Faders and you want the lighting values to persist after system shutdown/reboot without making changes to the show file itself.

13.1 The command set

The e:script language features a set of commands to write variables into an XML file for later use. At first, you have to use `XmlBegin` to specify a filename and if you want to read from or write into the particular file. The next step is to use the command `XmlExchange` to read/write certain variables (this depends on what you specified with `XmlBegin`). To "close" the file operation, call `XmlEnd`. You can use `XmlExchange` multiple times inside such an `XmlBegin`-/`XmlEnd`-block to read/write multiple variables from/into one file.

Please note that a write operation will completely erase the previous contents of the given file. So be careful to use the correct filename. If no file with the given name exists, a new file will be created.

In the following tables the you can find the complete syntax of the three commands:

```
XmlBegin
int XmlBegin(string filename,int mode);

filename – Filename
mode – 0 = Load XML from file, 1 = Save XML to file
return value: 1 for success, 0 otherwise
```

As a filename, you can either specify a simple filename (in that case the file will take its place in the Programmer's main directory) or a complete path along with the name of the file. Hint: To use the "My Documents" folder of the current Windows user you can write $(MyFiles) followed by subfolders (optionally) and the desired filename.

Important Note: When specifying a complete path, you must not use single backslashes `\` as a separator because these are interpreted as control characters. You can either use a double backslash `\\` or a single slash `/` as a separator.

```
XmlExchange
int XmlExchange(labelname variable);

variable – Name of existing variable
return value: 1 for success, 0 otherwise
```

Depending on the mode (read or write) that you specified with XmlBegin, the command XmlExchange either reads the value of the given name into the quoted variable or it writes the data from the variable into the file. In case you read out data from a file, the variable must as well exist in the file and of course, be already declared inside the macro. In the special case when you read out

an array, the array that was declared has also to be of exactly the same size as the array that was saved into the file before.

When you write a variable into a file, the particular variable must exist, of course.

> XmlEnd
> int XmlEnd();
>
> return value: 1 for success, 0 otherwise

The command `XmlEnd` "finishes" a read/write operation. Please note that a write operation will only occur when you use `XmlEnd`.

13.2 Example: Saving and Recalling Fader Values

Remember the example from the beginning of the chapter? To save the position of Action Pad faders we have to query the status of the associated Versatile Masters and save the particular values into a file. Assume that for our purpose it is sufficient to save the values of the first 12 Versatile Masters. The corresponding macro-code is very short and easy:

```
int vmaster[12];
int i;

for (i = 0; i < 12; i++)
{
            vmaster[i] = GetSpeedMaster(i);
}

XmlBegin("$(myfiles)\\vmasters.xml", 1);
XmlExchange(vmaster);
XmlEnd();
```

As you can see, an array named `vmaster` is declared in order to hold the values of the Versatile Masters. The values are received using the command `GetSpeedMaster` inside a for-loop. The contents of our array are then all together written into the file "vmasters.xml" inside the current users "My Documents"-directory.

What we have to do now is to write a macro that restores the Fader values from the XML file. This is quite as easy as our first macro and looks very similar to our previous code snippet:

```
int vmaster[12];
int i;

XmlBegin("$(myfiles)/vmasters.xml", 0);
XmlExchange(vmaster);
XmlEnd();

for (i = 0; i < 12; i++)
{
             SetSpeedMaster(i, vmaster[i]);
}
```

You see, we first read out the contents of our XML file (note the `0` as parameter for `XmlBegin`) into our array. These values are then applied to the particular Versatile Masters using the command

`SetSpeedMaster`.

13.3 Using a function for complex data exchange

For simple variable storage/recall operations involving only a few variables, the example above works just fine. However, if you want to exchange a big set of variables, maybe even several times, it comes in very handy to write a function for the data exchange operation. This simplifies the code and prevents errors like forgetting a variable in an exchange operation as well. The following macro code shows an example function exchanging some sample variables:

```
function DataExchange(int do_save)
{
            XmlBegin("vars.xml", do_save);
            XmlExchange(int1);
            XmlExchange(int2);
            XmlExchange(int3);
            XmlExchange(array1);
            XmlExchange(string1);
            XmlExchange(string2);
            XmlEnd();
}
```

The above function `DataExchange` has one parameter named `do_save`. This parameter specifies if you want to save or load the particular variables. As for both kinds of operation the same function is used, it can never occur that a variable is forgotten. To save the set of variables specified inside the function, call the function as follows:

```
DataExchange(1);
```

To restore a set of saved variables, write:

```
DataExchange(0);
```

14 Sending email from e:script

14.1 Background

Since early versions of the LAS it is possible to send email via e:script. Up to version 6.0 this email distribution was handled with an e:cue server as gateway. From LAS 6.0 SR1 on a direct SMTP interface is available to send mail.

First set the necessary parameters in the Application Options of the Programmer:

Name	**Set the sender's name for the FROM field.**
Email address	**The sender's email address.**
Server name	**The URL of the SMTP server**
Port	**The mail port of the server, depends on connection type and is set to values when the connection type is defined.**
Connection security	**The security type of the connection.**
Authentication method	**The encryption level for the connection.**
User name	**The user name used to login to the SMTP server.**
Password	**The password for the SMTP server login.**

14.2 Example

```
// Generate mail content
int hMailObject = CreateMail("user@example.com", "Hello World!",
"The quick brown fox jumps over the lazy dog.");
AddMailTo(hMailObject, "user@example.com");

// Schedule mail and remember the message queue ID
int nMessageQueueID = SendMail(hMailObject);
DestroyMail(hMailObject);

// Wait for message being processed
RegisterEvent(Frame, OnFrame);
Suspend();

function OnFrame()
{
            if (MailIsValidID(nMessageQueueID) == 0)
            {
                        alert(Unknown mail ID!\n");
                        exit; // shutdown suspended macro event
            }

            // Is message being processed?
            if (MailIsProcessed(nMessageQueueID) == 0)
            {
                        printf("Waiting for message being pro-
cessed ... \n");
            }
            else
```

```
            {
                        printf("Message was processed.\n");

                        // Is a response available?
                        if (MailIsResponseAvailable(nMessageQueu
eID))
                        {
                        // Check if sending was successful
                        if (MailIsTransmissionSuccessful
                                    (nMessageQueueID) == 1)
                              printf("Sending mail
successful.\n");
                        else
                              alert("Sending message
failed!\n");
                        }

                        exit; // Shutdown suspended macro event
            }
}
```

15 Peer connectors

15.1 Using peer connections

Using the “Peer Engine Connector” a PC running the e:cue Programmer can establish a communication channel to a Programmer running on another machine. This connection can handle different tasks that all will be discussed in this chapter:

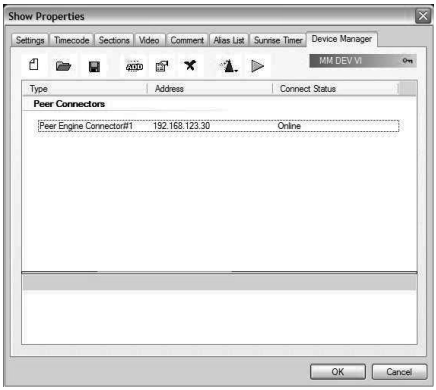
- Auto text redirection: Action Pad items can show auto text information delivered from another machine.
- Outputs and Inputs: Each Peer Connector can define up to 1024 output channels that will show up on the connected machine as inputs. Each input can be queried via e:script commands.
- ‘Exec Command’ redirection: Single macro commands can be redirected to a remote machine.
- ‘Call Macro’ redirection: Macro calls can be redirected to run a macro remotely.
- Peer Events: Peer events can set off trigger actions.

To open a connection, you have to add a Peer Engine Connector on both machines. This is done inside the Show Properties under ‘Device Manager’. On both machines, you have to specify the IP address of the computer that you want to connect to. For example, assume you have two PCs running with IP addresses `192.168.123.20` for PC one and `192.168.123.30` for PC two. On the first PC, add a Peer Engine Connector and in the field ‘IP Address’, write `192.168.123.30` as this is the PC we want to connect to. On the second PC, we go the other way round: We there need the address of the first PC, so we write `192.168.123.20`.

It is important to know that the Alias Names are the local names of the remote PC. The name will be used to address a certain peer, inside the e:script language and the Action Pad alike. We will go in more detail about that later on. The following picture shows the configuration dialogue for the first PC mentioned in the example.

In the picture, we have defined eight peer outputs. These will automatically show up as inputs on second PC when the connection has been established.

When you have finished the configuration, the new Peer Engine Connector should show up inside the Device Manager like this:



Note that there is no master/slave-logic behind these connections; communication takes place symmetrically and bidirectional with equal rights. Nevertheless, it is up to you what tasks the different peers handle.

15.2 Auto Text redirection

Although this topic is not directly related to e:script macro programming, it is a very interesting feature that is worth to be mentioned. We did already talk about the Action Pad Auto Text feature, which permits to display dynamic information on several Action Pad items. Since Programmer 3.8, which introduced the Peer Connector feature, it is possible to retrieve remote information from a connected peer machine. This can of course be set via macro language, just as you would do with 'local' Auto Text. But one after another: Let us first see how to redirect an Auto Text to retrieve information from a connected peer. The Syntax is simple; instead of just using the normal Auto Text syntax, you have to prefix the text in the following way:

```
\\peer#1\<cuelist 1 current>
```

You see, the right side is familiar to you as this is just normal Auto Text syntax. On the left side, starting with the two `\\`, we write the alias name of the peer from which we want to take the information. Right after the next backslash, the normal Auto Text Syntax starts, just like you know it.

This leads to very complex possibilities: With Auto Text, you can request the status of a connected device, e.g. a W+T Web-IO Digital, which has a set of inputs and outputs and is being connected via Ethernet. Combining this with a Peer Connector Auto Text, this makes it possible to gain information over two hops.

You see, in this example we have a W+T Web-IO Digital with a bunch of sensors on it. It is connected to a Media Engine 2 via Ethernet. The Media Engine is again connected to a second machine that is running an e:cue Programmer on it. Both have a connection established via Peer Connector. An Action Pad displaying Auto Text completes the cycle, showing the status of a certain sensor on the W+T Web-IO Digital. Sounds complicated, does it not? Well, when you have established all connections between the devices, the Auto Text itself will be easy. Here we go:

```
\\peer#1\<driver "wut#1" input 3 "ON" "OFF">
```

We address the certain peer that will, using Auto Text, display information about the connected device "wut#1", in detail, the status of input number three. Needless to say, we can use the same

syntax when we want to change Action Pad item text via e:script.

15.3 Inputs and outputs

We did already mention that each Peer Connector can define a number of outputs that will be visible as inputs on the connected machine. Via e:script, you can query available inputs as well as set certain values to the existing output channels. The input/output channels work with integer values. That means it can hold any number from -2.147.483.648 to 2.147.483.647.

To set a peer output to a certain value, the command `DriverSetOutputStatus` exists. Let us take a look at the syntax:

> DriverSetOutputStatus
> int DriverSetOutputStatus(int handle, int port_id, int value);
>
> handle - Device driver handle
> port_id - Zero based port number
> value – The value that shall be transmitted

We already know about the handle technique from the chapter about serial I/O. It works the same way here, use `DriverGetHandle` to receive the appropriate driver handle. The variable `port_id` defines the output port to which the value shall be sent. The following code example shows the process of setting a certain output to a desired value:

```
int handle;
handle = DriverGetHandle("peer#1");
DriverSetOutputStatus(handle, 0, GetCueCount(0));
```

Here, we set the first output (zero-based) of `"peer#1"` to the number of cues in the first cuelist. Once we have set it, the value is persistent until it is changed again.

To receive the current input value of a certain peer, we use the command `DriverGetInput-Statuset`. It works just like the command `DriverSetOutputStatus`, except for that there is no parameter `value`. The command's return value is nothing more than the value of the certain input port itself. The following code shows how to retrieve the value of the first input port coming from 'peer#1'. Assume this code is running on the machine to which we just sent the information from the last code snippet:

```
int handle, cue_id;
handle = DriverGetHandle("peer#1");
cue_id = DriverGetInputStatus(handle, 0);
printf("The peer#1 has currently %d cues in cuelist 1.\n", cue_
id);
```

We retrieve the input from the first port to 'peer#1', which (as we assume) has been set to the number of cues in the first cuelist of the remote machine. The "receiving" macro lacks practical use as long as we have to call it manually. Of course, we have several ways to add some automation to this process. How to add some automation will be discussed later in this chapter.

15.4 Sending key and init messages

A simple way to exchange information between peers without having to change the output ports is to use the command `SendKeyToPeer`. This will more or less simulate an e:com Terminal key press

and send it to just the stated peer. Furthermore, it will not activate Terminal Triggers because it has its own “Trigger class” (more about that can be found later in this chapter). The syntax is as follows:

```
int SendKeyToPeer(int handle, int key_id);
```

The parameter handle should be well known in the meantime; it will work here like everywhere else. The parameter `key_id` needs the id of the certain key that you want to send. The `key_ids` are similar to the ids sent when working with real e:com Terminals or the Terminal Emulator.

15.5 e:script redirection

You already know of some different methods to call a macro or to execute an e:script command. With Action Pad, Trigger and Macro Actions can do both. Another possibility is command execution via command line: On the lower-left side of the Programmer window, you find a small prompt window where you can execute macros and single e:script commands.

To execute a single e:script command via command line, just type in the command as you would in an e:script macro. The only difference is that you need to prefix the command with an exclamation mark ‘!’. This way, the Programmer knows that the following text is an e:script command. Example:

```
printf("Command Line execution has been successful!\n");
```

This will put the text “Command Line execution has been successful!” into the logbook.

To execute a macro, just type in the macro name, followed by up to five optional parameters separated with commas. Instead of using a ‘!’ as prefix, you will have to put a ‘$’ before your macro call. Example:

```
$CallMe, 4, 0, 7
```

This will execute a macro named “CallMe”, passing three parameters `4`, `0` and `7`.

The Peer Connector feature allows to remotely execute single e:script commands as well as complete macros on a connected machine. In case of macro execution, this does not mean that local macrocode will be transmitted to the remote machine; instead, the macro with the given name must already exist there. The syntax will look familiar to you:

```
!\\peer#1\printf("Remote e:script command execution has been
successful!\n);
```

will execute the `printf` command on the connected machine ‘peer#1’ while

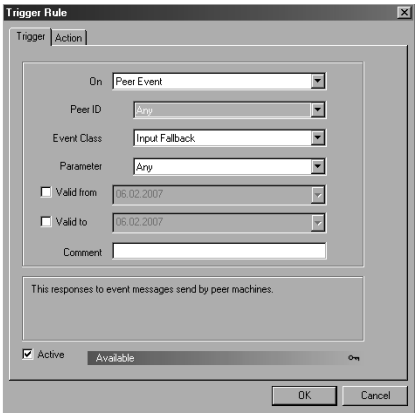
```
$\\peer#1\CallMe, 4, 0, 7;
```

will call the macro ‘CallMe’ via remote, again on ‘peer#1’.

Please note that both prefixes ‘!’ and ‘$’ are only necessary when you perform the commands via command line. If you let the remote commands be performed by any kind of Action (like a Trigger Event Action) you do not have to use them.

15.6 Peer events

The simplest way of automating the process of receiving peer inputs would be to call an appropriate macro repeatedly via cue action. To do so, record an empty cue into an arbitrary cuelist. Change the control mode to ‘wait’ and insert the wait time you want. Let the cue call your macro by creating a call action for this. When you now run the cuelist, your macro will be called repeatedly. This might not be the prettiest way to react on peer input, but it works and makes sense in certain cases, e.g. if there is a constant input flow between the peers to be worked on.



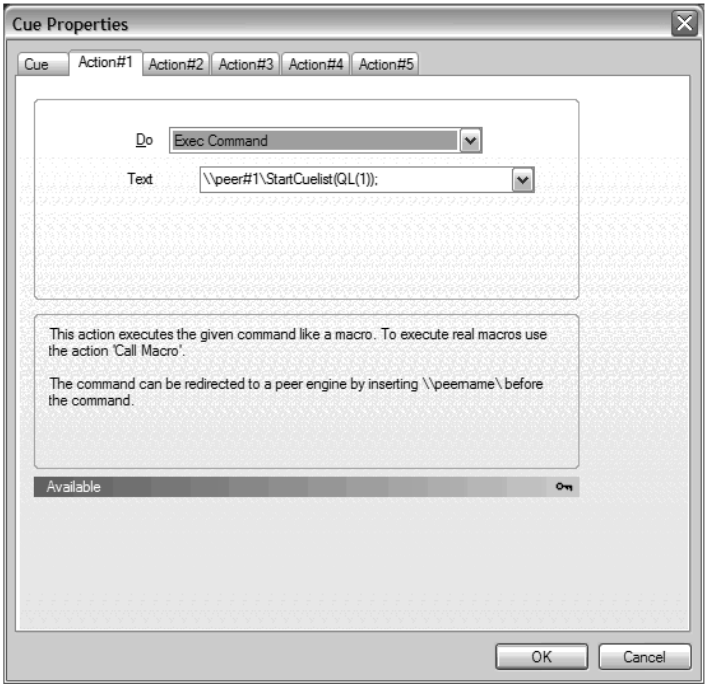
The other way of automatically reacting on peer input is a trigger rule. You already know about that from our chapter about serial input and it works just the same here. Just use “Peer Event” as a trigger and apply the correct Peed ID In this case, this is not the handle but the number behind the type specifier inside the Device Manager.

You have four different event classes to choose from. To react on changes on the input ports of a Peer Connector, use “Input Raise” and “Input Fallback”: An “Input Raise” will trigger whenever the input of the certain port (specified under ‘Parameter’) changes to a value not equal to zero while “Input Fallback” will trigger when a change to zero occurs.

The other two possible event classes will trigger whenever a peer key message and (respectively) a peer init message takes place.

15.7 Example: controlling cuelists on a peer machine

A very handy example for using Peer Connectors is remote controlling the cuelists present on a connected PC. This is a very easy task: On your local PC, create a cue and as a cue action, set “Exec Command”. Inside the box ‘Text’, you use the e:script command `CuelistStart`, with the desired cuelist number as the single parameter. This command, of course, has to be redirected to the desired Peer, so it has to be prefixed with the familiar syntax: `\\peer#1\CuelistStart(QL(1))` will start the first cuelist on the remote PC.



Analog, you can stop the cuelist using `CuelistStop` instead: `\\peer#1\CuelistStop(QL(1))` will stop the first Cuelist on the remote PC.

You could even start or stop a great number of Cuelists by making use of a loop:

```
\\peer#1\int i; for (i=0;  i<50;  ++i) CuelistStart(QL(100+i));
```

and equivalently

```
\\peer#1\int i; for (i=0;  i<50;  ++i) CuelistStop(QL(100+i));
```

will start/stop the Cuelists from 100 to 150.

16 I/O boxes

The e:cue Programmer features support for several I/O Boxes from different manufacturers. Drivers exist for e:cue's Moxa ioLogik series, the W&T Web-IO Digital and the Kiss-Box DI8 DO4. Additionally, the e:cue Media Engine 2 Input Module, which works the same way, can be used.

The basic functionality of these boxes is in all cases the same, differing only in detail. The box has to be connected via Ethernet, with an own IP assigned. A box has several input and output ports, in most cases digital, in some cases analog. The output ports can be set to certain values while the inputs can be read out. For more Details about this, please refer to the documentation of the particular box that you want to use.

The e:cue Programmer's driver model allows all boxes to be accessed mainly the same way, using the same e:script commands. This allows us to give a common introduction of basic I/O access via e:script without having to go into detail for specific boxes. Nevertheless, there are several syntactic differences Between the different boxes. Here it will be assumed that a Moxa E2210 I/O Box is connected and configured properly.

What to do with the inputs and outputs depends completely on you and the devices connected to the I/O Box, respectively. Possible uses might be to switch on/off devices, get status feedback or communicate with a device for configuration or control tasks. The following section will give an overview about the different supported I/O Boxes along with detailed configuration instructions.

16.1 Driver model interfaces

To access IO boxes there is a generalized command set in e:script. These commands are supported for all IO devices.

```
int DriverInputGetCount(int handle);
int DriverInputGet(int handle, int port_id);
int DriverOutputGetCount(int handle);
int DriverOutputGet(int handle, int port_id);
int DriverOutputSet(int handle, int port_id, int set);
int DriverPropertyGetInt(int handle, string prop_name,
                         int-ref value);
int DriverPropertyGetString(int handle, string prop_name,
                         string-ref value);
int DriverSendCommand(int handle, string command, string param1,
                         int param2);
```

16.2 Querying status information

When we were dealing with device drivers, we always had to obtain a driver handle for the certain device. When working with I/O Boxes, there is no difference with that. Whenever you want to address an I/O box, the first thing to do is to use the command GetDriverHandle with the device alias name as parameter. Though you should already be familiar with that, it can be seen here again:

```
int handle;
handle =  DriverGetHandle("term#1");
```

This code snippet finds out the driver handle of the device with alias name "term#1" and saves it to the variable `handle`.

To query the status of single input/output ports, we have two possibilities: The commands `Driver-`

`InputGet` and `DriverOutputGet` both need the driver handle and the designated input/output port as parameters. Both commands will deliver the current value of the given port, zero or one, as an integer. As an example, we will query all output ports of our I/O box. To acquire the port count, we use the command `DriverOutputGetCount` (an equivalent command `DriverInputGetCount` also exists, of course):

```
int handle, count, i;
handle = GetDriverHandle("term#1");
count = DriverOutputGetCount(handle);

for (i = 0; i < count; i++)
            printf("Status of Port %d is: %d\n", i,
DriverOutputGet(handle, i));
```

The other possibility is to use the commands `DriverPropertyGetString` or `DriverPropertyGetInt` with an appropriate property parameter. Let us look at the syntax: `DriverPropertyGetInt`

Device drivers can return different status values reflecting the status of the connected device.

> int DriverPropertyGetInt(int handle, string prop_name, int-ref value);
>
> Parameters:
> handle - Device driver handle.
> prop_name - The name of the property.
> value - The property return value.
>
> Return Value:
> 1 for success, 0 otherwise.

The properties themselves differ from device to device. For our example with the Moxa E2210, the correct property name to query a single output port would be `DIGITAL_OUTPUT_x`, where `x` stands for the desired port (zero based). The parameter `value` will receive the particular property status, in integer format. The command `DriverPropertyGetInt` works the same way, but instead of having an integer reference parameter, you have to pass a string which then will receive the property status in string format.

To get the status of a certain output port use these functions:

```
int handle, count, i, value;
string prop_name;

handle = DriverGetHandle("term#1");
count = DriverOutputGetCount(handle);
for (i = 0; i < count; i++)
{
            prop_name = format("OUTPUT_%d", i);
            DriverPropertyGetInt(handle, prop_name, value);
            printf("Status of Port %d is: %d\n", i, value);
}
```

In this case, this is more complicated than using `DriverOutputGet`. But with another property, we can make this a lot easier (this is device dependant, but for most boxes, equivalent properties exist). You can pass the property OUTPUT_ALL to receive the status of all output ports at once. When

using `DriverPropertyGetInt`, the status will be returned as a single integer, where each single bit represents the status of a single port. To extract the information, the simplest approach is to use bit operators (more about that can be found in chapter 1.8.4). When using `DriverPropertyGetString`, a string will be returned where each char holds either a zero or a one, again to represent the status of the single ports. In the following code examples, we will see both commands, as well as an attempt to “decode” the input:

```
int handle, count, i, int_val;
string prop_name;

handle = DriverGetHandle(“term#1”);
count = DriverOutputGetCount(handle);
DriverPropertyGetInt(handle, “OUTPUT_ALL”, int_val);
for (i = 0; i < count; i++)
{
            printf(“Status of Port %d is %d\n”, i,
                        (int_val & (1 << i)) >> i);
}
```

Most of this code should be well known, but the code inside the for loop needs further explanation, especially `(int_val & (1 << i)) >> i`. We are going to work through the brackets from inside to the outside. With `1 << i`, we create the “ith” power of 2. To understand this, think of the binary representation of 2 to the power of i. This is always the ith bit set to one, while the others are set to zero. To find out if the ith bit of our status variable int_val is set to one, we now only have to make a bitwise AND-interconnection with the number we just created and shift the result back to the right by i bits. The best way to understand this is to write down a simple example for a fixed input value and a fixed i:

Assume the input value is 164 (Binary: 1010 0100) and we want to analyze the 3rd bit. The bitwise And-interconnection will then be 10100100 AND 0000 0100, which results in 0000 0100. This, shifted back by three bits to the right, results in 1, which means that the status of the 3rd bit is 1.

Now let us watch the equivalent code for a status output of type string:

```
int handle, count, i, spacecount;
string prop_name, str_val;

handle = DriverGetHandle(“term#1”);
count = DriverOutputGetCount(handle);

DriverPropertyGetString(handle, “OUTPUT_ALL”, str_val);
printf(“ %s\n”, str_val);
for (i = 0; i < strlen(str_val); i++)
{
            if (strcmp(midstr(str_val, i, 1), “ “) == 0)
                        spacecount++;
            else

                        printf(“Status of Port %d is %d\n”,
                         i - spacecount,val(midstr(str_val, i,
1)));
}
```

What is mainly used here is some string intersection and conversion, but there is a little rub in it, what makes everything a bit more complicated: Every four chars, the string of numbers is separated by a single space. Due to this, it is not possible to directly map the ith character to the ith port because this will be wrong after the first space occurred. To handle this, we let the loop run through the complete string and every time we find a space, we increase a designated counter variable. This counter now has just to be subtracted from the current value of `i` to get the appropriate port number.

16.3 Changing values of the output ports

Working with an I/O Box would not be quite as useful without being able to change the output ports from within the Programmer. Like when we were querying the I/O-ports, again we have two possibilities.

The simplest approach is using the command `DriverOutputSet`. The syntax is very similar to the commands `DriverOutputGet` and `DriverInputGet`, with the difference that you have to pass an output value for the certain port as well:

```
int DriverOutputSet(int handle,int port_id,int set);

Parameters:
handle - Device driver handle.
port_id - Zero based port number
set - 1 = activate the output, 0 = deactivate    the output

Return Value:
1 for success, 0 otherwise
```

As you can see, there is nothing big about it. To set a specific output port to “on”, pass a one to the `set` parameter, while passing a zero will set the port to “off”.

A practical use might be that you have an action pad with eight buttons, each calling the same macro with one parameter passed. The parameter should be the zero based port number so that the macro knows which port to deal with. The macro itself will check which value the certain output port has and then invert that value:

```
int port, handle, status;

port = getarg(0);
handle = DriverGetHandle("term#1");

status = DriverOutputGet(handle, port);
DriverOutputSet(handle, port, status ^ 1);
```

The variable `status` holds the current status of the certain port. To shorten the e:script a bit, the inverting of the variable status is done directly inside the command `DriverOutputSet`. It is done using a bitwise XOR with 1.

The second possibility to change the values of certain output ports is the use of Device Commands. The command `DriverSendCommand` sends command strings followed by context dependend parameters to the box. Let us look at the command definition:

int DriverSendCommand(int handle,string command,
string param1,int param2);

Parameters:
handle - Device driver handle.
command - Primary command string
param1 - First parameter for given command or empty string, depending on command type.
param2 - Secondary numerical parameter. Use zero if the command requires no secondary parameter.

Return Value:
1 for success, 0 otherwise

The command string is very similar to the principle of a property string that you got to know in the last chapter. For our intention to change the status of certain output ports, we will make use of the command strings OUTPUT_SET_ON and OUTPUT_SET_OFF. The appropriate `param1` for these are PORT_x (where x defines the port number, zero based) and PORT_ALL (this will address all ports at the same time). For both command strings, `param2` is not needed, so it has just to be set to zero. To show some practical use of it, we will take on the example above and this time implement our macro by using `DriverSendCommand` instead of `DriverOutputSet`:

```
int port, handle, status;
string info;

port = getarg(0);
info = format("PORT_%d", port);
handle = DriverGetHandle("term#1");
status = DriverOutputGet(handle, port);

if (status)
{
            DriverSendCommand(handle, "OUTPUT_SET_OFF", info, 0);
}
else
{
            DriverSendCommand(handle, "OUTPUT_SET_ON", info, 0);
}
```

This nearly works the same way as our previous macro, with the only difference that we have to build up a parameter string to specify the port, and that we cannot use an XOR to elegantly invert the port status.

16.4 Device setup and access

General device setup

The device-specific adjustments are made in the “Device Setup” from the Device Manager in the Programmer (see the System Manual for the LAS for details). The main setting is the alias name. Please note that this alias name must be unique! You use this name to access a device in the automation, trigger list, e:script or Action Pad.

Device properties

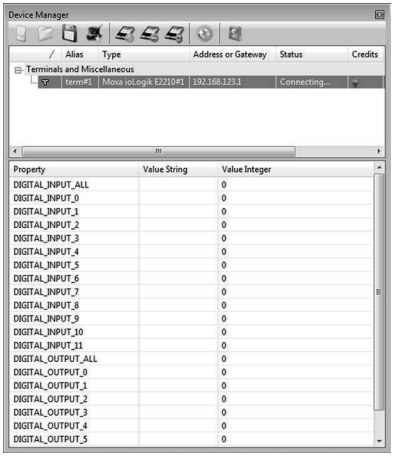
Special characteristics of the driver, like the condition of an input and/or an output are e.g. represented in the Device Properties dialog. You can query these Device Properties via e:script instructions.

Always, at first, an IP address and subnet mask must be assigned to the devices. This can be done using the provided Setup and Configuration Applications for the boxes.. For use with the Programmer it must be guaranteed that the box has no password protection, so you have to deactivate any password protection.

Device command

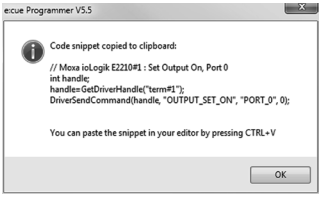
“Device Commands” can call special functions of the connected equipment. The simplest way of a call is the use of the context menu in the device manager. Clicking on the driver entry with the right mouse button will open the menu.

The status of the device becomes visible in the Device manager as soon as you click to select an entry.



e:script

The Device Commands can be called via e:script. Select a device, press the right mouse button and select Create e:script. In this context menu select the port to use and the desired state. The e:script code snippet will be copied to the Windows Clipboard, insert the code in your macro using Ctrl-V.



16.5 Code example

```
// Moxa ioLogik E2210#1 : Set Output Off, PORT_6
int handle;

handle=DriverGetHandle(“term#1”);
DriverSendCommand(handle, “OUTPUT_SET_OFF”, “PORT_6”, 0);

// Print the Input States
```

```
int port_id;
int handle;
handle=DriverGetHandle("term#1");
for (port_id=0; port_id < DriverInputGetCount(handle); port_id++)
{
  printf("Input %i: %d\n", port_id, DriverInputGet(handle,
                           port_id));
}


// Set all the Output States on false
int port_id;
int handle;
handle = DriverGetHandle("term#1");
for (port_id=0; port_id < DriverOutputGetCount(handle); port_id++)
{
  DriverOutputSet(handle, port_id, 0));  // 0=false, 1=true
}
// Print the Output States
int port_id;
int handle;

handle = DriverGetHandle("term#1");
for (port_id=0; port_id < DriverOutputGetCount(handle); port_id++)
{
  printf("Output %i: %d\n", port_id, DriverOutputGet(handle,
                           port_id));
}

// Execute a Device Command, Port 0 will be set to on state
int handle;

handle=DriverGetHandle("term#1");
DriverSendCommand(handle, "OUTPUT_SET_ON", "PORT_0", 0);

// Get a Device Property, all Input values
string value;
int handle;
handle=DriverGetHandle("term#1");
DriverPropertyGetString(handle, "DIGITAL_INPUT_ALL", value);
printf("DIGITAL_INPUT_ALL: %s", value);
```

16.6 Moxa ioLogic E22xx and E12xx

Since LAS 5.5 SR1 the models E1210, E1211, E1212, E1214, E1240, E2210 and E2240 are supported. Also a freely configurable TCP/IP driver is available which supports digital and analog inputs and outputs.

The Automatic Setup Device Wizard of the Programmer supports all seven Modbus Devices. For the discovery the correct network card must be defined under Application Options | Advanced | Moxa. Also, with LAS 5.5 SR1, the property names of the inputs and outputs have

changed for the E2210 and E2240. The inputs and outputs changed from INPUT_0 to DIGITAL_INPUT_0 or to ANALOG_INPUT_0, depending on the input or output of the device. This was necessary because a Modbus driver can have both kinds of inputs and output.

This affects the two e:script methods, which have to be adjusted when updating to SR1:

```
DriverGetPropertyStringValue( )
DriverGetPropertyValue( )
```

At first, an IP address and subnet mask must be assigned to the MOXA IO boxes. This can be done for 224x using the provided MOXA IOAdmin, for 221x the iOSearch configuration software. For use with the Programmer it must be guaranteed that the box has no password protection, so you have to deactivate any password protection.

The inputs must be configured as [DI] and the outputs as [DO]. The modes [EventCounter] and [PULSE] are not supported by the Programmer

Device setup

Property	Value	Description
Alias Name	term#1	A unique device name. You can access this driver using the alias name. It will be used inside the e:script and the event handler.
IP Address	x.x.x.x	The IP address of the Moxa Box.
Refresh Interval Time	x	An interval time in milliseconds, in which regularly the box is queried. The minimum interval time is about 1000 ms. The lower this value is, the more resources the system will use.
Send Trigger Events	on/off	Determines whether a KeyPress and/or a KeyRelease event is sent upon change of the input signal. The processing of trigger events does consume system resources.
Output Init Mode	Default…/ Maintain Value…	With “Default…” the outputs are initialized with the defaults set under “default output [DO] 0..7”. When using “Maintain…” the current value of the box is used upon initialization.
Default Output Value [DO] 0..7	00 to FF	Init value of the Outputs in hexadecimal notation. This value will only be set when you set the “Output Init Mode” to “Default Output Value”.

Device commands

Command	Param	Description
DIGITAL_INPUT_ALL	0000 0000 0000	All states of the Input Channels as a binary string. each number stands for a single input channel.
DIGITAL_OUTPUT_ALL	0000 0000	All states of the output channels 0..7 as a binary string. Each number stands for a single output channel

DIGITAL_INPUT_x	true/false, 0/1	State of the Input Channel x from 0 to 11
DIGITAL_OUTPUT_x		State of the Output Channel x from 0 to 7
ANALOG_INPUT_x	**-32768 to 32767**	**All values of the Input Channels 0..7**
ANALOG_OUTPUT_x	**0 to 4095**	**All values of the Output Channels 0..1**
Command Name	Param	Description
OUTPUT_SET_VALUE	**PORT_x VALUE**	**Set the Output Channel No x [0..7] to a specific value.**

Example:

Set the output 0 to the maximum value possible.

```
// Moxa ioLogik E2240#1 : Set Output Value, PORT_0
int handle;
int value;

value = 4095;              // 0..4095
handle=DriverGetHandle("term#1");
DriverSendCommand(handle, "OUTPUT_SET_VALUE ", "PORT_0", value);
```

16.7 W&T Web I/O 12xDigital-IO

Device Setup

Property	Value	Description
Alias Name	term#1	A unique device name. You can access this driver using the alias name. It will be used inside the e:script and the event handler.
IP Address	x.x.x.x	The IP address of the W&T Web IO box
Refresh Interval Time	x	An interval time in milliseconds, in which regularly the box is queried. The minimum interval time is about 1000 ms. The lower this value is, the more resources the system will use.
Send Trigger Events	on/off	Determines whether a KeyPress and/or a Key-Release event is sent upon change of the input of signal. The processing of trigger events does consume system resources.
Web IO Type	12x Digital	Device Type. At this moment there is only support for the 12x Digital Box.
Output Init Mode	Default…/ Maintain Value…	With “Default…” the outputs are initialized with the defaults set under “default output [DO] 0..7”. When using “Maintain…” the current value of the box is used upon initialization.
Default Output Value	0000 bis FFFF	Init value of the Outputs in hexadecimal notation. This value will only be set when you set the “Output Init Mode” to “Default Output Value”.

Device commands

Command	Param	Description

INPUT_ALL	0000 0000 0000	All states of the Input Channels 0..11 as a binary string. Each number represents a single input channel.
OUTPUT_ALL	0000 0000 0000	All states of the output channels 0..7 as a binary string. Each number represents a single output channel.
INPUT_x	true/false, 0/1	State of the Input Channel x from 0 to 11.
OUTPUT_x		State of the Output Channel x from 0 to 1.
OUTPUT_SET_ON	PORT_x	Set the Output Channel No x [0..11] on true.
OUTPUT_SET_ON	PORT_ALL	Set all Output Channels No [0..11] on true
OUTPUT_SET_OFF	PORT_x	Set the Output Channel No x [0..11] on false.
OUTPUT_SET_OFF	PORT_ALL	Set all Output Channels No x[0..11] on false

16.8 Kiss Box DI8 DO4

Device Setup

Property	Value	Description
Alias Name	term#1	A unique device name. You can access this driver using the alias name. It will be used inside the e:script and the event handler.
IP Address	x.x.x.x	The ip address of the KISS box.
Port Send	9813	The KISS box sends the UDP packages to the chosen destination port.
Port Listen	9817	The programmer listens for UDP packages from the KISS box at this port.
Refresh Interval Time	x	An interval time in milliseconds, in which regularly the box is queried. The minimum interval time is about 1000 ms. The lower this value is, the more resources the system will use.
Send Trigger Events	on/off	Determines whether a KeyPress and/or a KeyRelease event is sent upon change of the input of signal. The processing of trigger events does consume system resources.

Slot #x Type
x = 1..8
	Disabled
	Input DI8-DC
	Output DO4-S
		Depending on design, the Kiss box offers up to eight Slots. Each Slot can either be “Disabled”, or equipped with an “Input DI8-DC” or an “Output DO4-S” - module.

Slot #xDefault Output Value x x= 1..8	0 bis F	If a slot is equipped with an “Output DO4-S” – module, you can define an init value here. The init value is written in hexadecimal notation.

Device commands

Command	Param	Description
INPUT_ALL_x	0000 0000	All states of the Input Channels 1...8 of the slot x from 1 to 8 as a binary string. Each number represents a single input channel

OUTPUT_ALL_x	0000	All states of the Input Channels 1...4 of the slot x 1...8 as binary string. Each number represents a single input channel.
INPUT_x_y	true/false, 0/1	State of the Input Channel y from 1 to 8 of the slot x from 1 to 8
OUTPUT_x_y		State of the Output Channel y from 1 to 4 of the slot x from 1 to 8
OUTPUT_SET_ON_x	PORT_y	Set the Output Channel No y [1..4] of the slot x to true.
OUTPUT_SET_ON_x	PORT_ALL	Set all Output Channel of the slot x to true.
OUTPUT_SET_OFF_x	PORT_x	Set the Output Channel No y [1..4] of the slot x to false.
OUTPUT_SET_OFF_x	PORT_ALL	Set all Output Channel of the slot x to false.

Example

```
// KISS-BOX DI8 DO4#1 : Set Output On Slot #2, PORT_4
int handle;

handle=GetDriverHandle("term#1");
DriverSendCommand(handle, "OUTPUT_SET_ON_2", "PORT_4", 0);
```

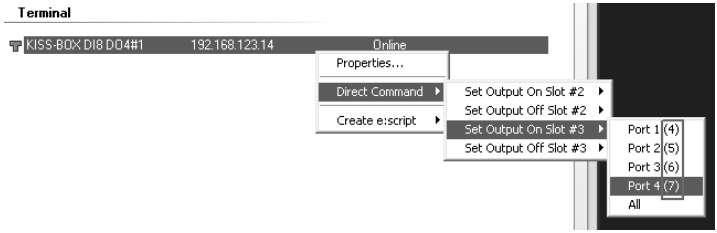
The e: script instructions `DriverInputGet`, `DriverOutputGet`, `DriverOutputSet` all use `port_id` as parameter. Since the Kiss box is arranged very flexible concerning its in- and outputs, the following rule applies for port_id definition:

Inputs

The first input module with begins with `port_id = 0` and ends with `port_id = 7`. The next would begin with 8 and end with 15 etc.

Outputs

The first output module with begins with `port_id = 0` and ends with `port_id = 3`. The next would begin with 4 and end with 7 etc. Look at the picture below. The ids in the brackets are valid numbers for `port_id`.



16.9 Projection Devices

PJLink Projector

Features

9 x RGB

9 x Video
9 x Digital
9 x Storage
9 x Network

PJLink is a protocol based on TCP/IP for control of projector devices, developed by a Japanese consortium and used by various Japanese manufacturers: http://pjlink.jbmia.or.jp.

At first, an IP address and subnet mask must be assigned to the projector. In most cases, this is possible with a web interface. For further information, take a look at the projector documentation. Be sure that your projector does not use any kind of password protection.

Device setup

Property	Value	Description
Alias Name	term#1	A unique device name. You can access this driver using the alias name. It will be used inside the e:script and the event handler.
IP Address	x.x.x.x	The IP address of the PJLink projector.
Refresh Interval Time	x	An interval time in milliseconds, in which regularly the box is queried. The minimum interval time is about 1000 ms. The lower this value is, the more resources the system will use.

Device commands

Command	Param	Description
POWER	ON OFF	
INPUT	RGB_x [1..9] VIDEO_x [1..9] DIGITAL_x [1..9] STORAGE_x [1..9] NETWORK_x [1..9]	
MUTE_AUDIO_VIDEO	MUTE_AUDIO_VIDEO VIDEO_ON AUDIO_OFF AUDIO_ON AUDIO_VIDEO_OFF AUDIO_VIDEO_ON	

16.10 Panasonic PTD

This driver can control various Panasonic projectors via RS-232. The driver was written for and has been tested with the Panasonic PT-D3500E. According to manufacturer specifications, it should also work with the devices PT-D7700/DW7000, PT-D5600/DW5000 and PT-D5500. These models have not been tested! See www.panasonic.com

Hardware configuration

The projector must be connected to a valid COM port (RS-232). The COM port interface parameters must be configured inside the projector and the Programmer as well.

Device setup

Property	Value	Description
Alias Name	term#1	A unique device name. You can access this driver using the alias name. It will be used inside the e:script and the event handler.
Serial Port	COM x	The IP address of the Panasonic PTD.
Refresh Interval Time	x	An interval time in milliseconds, in which regularly the box is queried. The minimum interval time is about 1000 ms. The lower this value is, the more resources the system will use.

Device commands

Command	Param	Description
POWER	ON OFF	Power on/off
INPUT	RGB_x [1..3] VIDEO_1 SVIDEO_1 DVI_1 AUX_x [1..3]	Video input source
SHUTTER	ON OFF	Shutter on/off
FREEZE	ON' OFF	Freeze the image
AUTOSETUP		
MUTE_AUDIO_VIDEO	MUTE_AUDIO_VIDEO VIDEO_ON AUDIO_OFF AUDIO_ON AUDIO_VIDEO_OFF AUDIO_VIDEO_ON	
TEST	OFF WHITE BLACK CHECKER_1 CHECKER_2 WINDOW_WHITE WINDOW_BLACK CROSS_HATCH COLOR_BAR RAMP	Test images
OSD	ON OFF	On Screen Display on/off

PICTURE_MODE	NATURAL STANDARD DYNAMIC CINEMA GRAPHIC	Image manipulation
COLOR_TEMP	LOW MIDDLE HIGH DYNAMIC USER_1 USER_2 STANDARD NATURAL CINEMA GRAPHIC DEFAULT	Color temperature
COLOR_INTEN-SITY	PERCENT_x VALUE [0..63]	Color intensity
CONTRAST	PERCENT_x VALUE [0..63]	Contrast
BRIGHTNESS	PERCENT_x VALUE [0..63]	Brightness
VOLUME	PERCENT_x VALUE [0..63]	Volume
TINT	PERCENT_x VALUE [0..63]	

Supported e:script commands

```
int DriverPropertyGetInt(int handle,
                         string prop_name,int-ref value);
int DriverPropertyGetString(int handle,string prop_name,
                         string-ref value);
int DriverSendCommand(int handle,string command,
                         string param1,int param2);
```

With dynamic value

```
// Panasonic PTD RS232#1 : Color Intensity, 0..63
int handle;
handle = DriverGetHandle("term#1");
int value;
value = 0; // 0...63
DriverSendCommand(handle, "COLOR_INTENSITY", "VALUE", value);
```

With fixed value

```
// Panasonic PTD RS232#1 : Color Intensity, 100%
int handle;
```

```
handle = DriverGetHandle("term#1");
DriverSendCommand(handle, "COLOR_INTENSITY", "PERCENT_100", 0);
```

17 HTTP client

17.1 Background

With the new HTTP client driver you can connect the Programmer to HTTP servers and get pages or update a status on them using the HTTP commands GET/POST. In the Add Driver Dialog inside the Device Manager add the HTTP client driver:

Beside the standard properties you can configure the server address you want to connect to.

In e:script you can use the driver handle to send requests (POST/GET or other) and receive responses using the following commands:

HttpRequestCreate
HttpRequestCreateFrom
HttpRequestDestroy
HttpRequestSend
HttpResponseGet
HttpResponseGetAndWait

Define the source server in the driver definition and use a relative path in the request.

17.2 Example

```
if (!exists(_resp)){
            int _resp = -1;
}

int bobg;
int drv = DriverGetHandle("protocol#1");
int request = HttpRequestCreate("GET","/gettext.php","");

_resp = HttpRequestSend(drv,request);
HttpRequestDestroy(request);
RegisterEvent(HTTPResponseEvent, OnHTTPResponseEvent);
Suspend();

function OnHTTPResponseEvent(int p0,int p1,int p2) {
            if(_resp == p0) {
            bobg = HttpResponseGet(DriverGetHandle("protocol#1"),
p0);
            } else {
            alert("_resp != p0!!!\n");
            }
            printf("%s",BobGetString(bobg,0,BobGetSize(bobg)));
            exit;
}
```

18 UDP communication

It is possible to send UDP packets from e:script macros for direct communication to devices.

```
RegisterEvent(UdpReceive, OnUdp);

int nBob;
nBob = BobAllocate(1500);
int nLen;
int nPort;
string sIpAddress;
string sText;

Suspend();

function OnUdp(int nDriverHandle)
{
            nLen = ReceiveFrom(nDriverHandle, nBob, sIpAddress,
nPort);
            if (nLen <= 0)
            {
                        return;
            }

            sText = BobGetString(nBob, 0, 255);
            printf("Incoming Message: %s\n", sText);
}
```

This macro is very nice, but also completely useless without any sender who can deliver content. As another proof of concept and to show how sending UDP data works, we will now create a simple macro to send strings using UDP.

18.1 Sending UDP packets

Most that we need is already well known from the previous topic, so this is very easy. Once again we need a Bob. In this case, we have to write the data that we want to send into the Bob using BobSetString. And then use the command SendTo to send out the data to an arbitrary IP address and port number. We also must retrieve the UDP driver handle manually as this time we do not get it passed as an event function parameter. Here is the corresponding macro code:

```
int nBob;
nBob = BobAllocate(1500);

int nHandle;
nHandle = DriverGetHandle("protocol#1");

string sMessage;
sMessage = "UDP Send and Receive Test!\n";
BobSetString(nBob, 0,255, sMessage);
SendTo(nHandle, nBob, 255, "172.22.122.92", 8080);
```

The command BobSetString works quite similar to the BobGetString command. You pass the Bob handle, an offset, length and the message itself and this construct will put the text into the Bob. Then you use the command SendTo to send out the data. The parameters are the UDP driver handle, the Bob handle, length of the data that is to be sent, IP address (as string) and a port number.

You can use these both macros on one machine and simply send to your computer's IP address and the port of the UDP driver. Then the server macro will automatically receive the messages you send.

19 TCP client driver

In addition to the already existing UDP driver from LAS 6.0 now has a TCP Client driver available. This driver allows you to establish a TCP connection with a server and send or receive packets to/ from it.

19.1 Receiving TCP packets

The logic behind the example is similar to the UDP example.

```
// TCP receive script
RegisterEvent(TcpReceive, OnTcpReceive);
string sText;
int bytesReceived;

Suspend();

function OnTcpReceive(int driverHandle) // received TCP frame
{
            int bobHandle = BobAllocate(4096);
            bytesReceived = TcpReceive(driverHandle, bobHandle);
            if (bytesReceived > 0)
            {
                        sText = BobGetString(bobHandle, 0,
bytesReceived + 1);
                        printf("Received %d bytes.\nData: %s\n",
                                    bytesReceived,
sText);
            }
            BobFree(bobHandle);
}
```

The related TCP commands are TcpSend() and TcpReceive(). See the crossreference in the Programmer for details.

20 The e:net protocol

20.1 Introduction

The totality of all protocols used by devices of the e:cue product series is called e:net. Both the e:cue Media Engine and the e:cue Programmer are able to communicate with connected terminals using UDP messages. The whole communication takes place in small standard UDP packets. Normally these packets are broadcasted but they can also be sent to a certain IP. In most cases an acknowledgement of the receiver does not occur.

All given code samples are written in C++ programming language.

The general structure of an `e:net` packet is as follows:

```
#pragma pack(1)
typedef struct
{
            DWORD          magic;
            DWORD          static_key;
            DWORD          device_status;
            DWORD          device_type;
            DWORD          device_version;
            DWORD          event_id;
            DWORD          event_par;
            char           filename[256];
            char           extra[249];
            char           link_status;
            UCHAR          mac[6];
} ECUE_BROADCAST;
#pragma pack()
```

Variables of type `DWORD` are defined as 32-bit unsigned int. In other development environments or programming languages, where `DWORD` is not available, it is important to use equivalent types.

20.2 General information

The variable `magic` must be assigned with the constant `MAGIC_ECUE`. This constant defines as follows:

```
#define MAGIC_ECUE          0x45435545
```

This is the hexadecimal interpretation of the chars `ECUE`. The preprocessor-command

```
#pragma pack(1)
```

defines the structure as ‘packed’. That is the compiler must not insert empty blocks on its part to align single variables to DWORD borders. If you use development environments other than Microsoft Visual Studio, you may have to adjust the command or use an equivalent one.

All messages of type `ECUE_BROADCAST` are sent via Port 31772.

The Programmer and the Media Engine have to receive all variables Intel-typical in little endian format. So if you implement any e:net messaging on a machine using big endian memory addressing (like PowerPC) or if you want to implement the protocol in Java (which also uses big endian format), it is essential to have the data arranged correctly.

The MAC address does not necessarily need to be specified. It is sufficient to assign a zero into

the particular variable instead.

The following hex dump is a UDP packet recorded with the network protocol analyzer software Wireshark (UDP header are the first three lines):

```
0000   ff ff ff ff ff ff 00 e0 4c 7c 89 7e 08 00 45 00  ........L|.~..E.
0010   02 38 44 8b 00 00 80 11 b8 6c c0 a8 7b 15 ff ff  .8D......l..{...
0020   ff ff 7c 1d 7c 1c 02 24 33 13 45 43 55 45 00 00  ..|.|..$3.ECUE..
0030   00 00 00 00 00 00 03 00 00 00 00 00 00 00 02 00  ................
0040   00 00 07 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0060   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0070   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0080   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0090   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00a0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00b0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00c0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00d0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00e0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00f0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0100   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0110   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0120   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0130   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0140   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0150   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0160   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0170   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0180   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0190   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01a0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01b0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01c0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01d0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01e0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01f0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0200   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0210   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0220   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0230   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
0240   00 00 00 00 00 00                                ......
```

All data inside the packet is ordered big endian because the UDP protocol will always arrange data this way. You can see this best at position `0x2A`, where the variable `magic` is positioned. As mentioned before, this variable is always defined as “ECUE” (ASCII) or `0x45435545` but in the UDP dump, it appears as “ECUE” or `0x45554345`.

20.3 Sending a status broadcast

All e:net devices continuously send broadcast messages into the net to indicate their presence. For that purpose a message of type `ECUE_BROADCAST` is sent to the Ethernet. A status message should be sent at least every 3 seconds but should also not be sent significantly more often to avoid unnecessary network traffic. A device that shows the active e:net devices in a list should use a timeout of 10 seconds. A good example for a device list is found in the e:cue Programmer status window under ‘Visible Network Devices’.

For a status broadcast, the single struct members will receive the following values:

magic	MAGIC_ECUE
static_key	0
device_status	0
device_type	3 = e:com Terminal
device_version	0
event_id	0 = no event
event_par	0
filename	0
extra	0
link_status	1
mac	the sender's mac-address

20.4 Sending a keyboard event

A keyboard event occurs e.g. if any key on an e:com Terminal has been pressed or released. The e:com Terminal will send this event as a broadcast to the network.

Other Systems can easily trigger e:cue System events by simulating e:com keyboard events. On side of the e:cue system, keyboard events can directly be associated to actions or be further processed using macros.

magic	MAGIC_ECUE
static_key	0
device_status	bitmask with current keyboard status
device_type	3
device_version	0
event_id	2 = a key has been pressed
event_par	key number
filename	0
extra	0
link_status	1
mac	the sender's mac-address

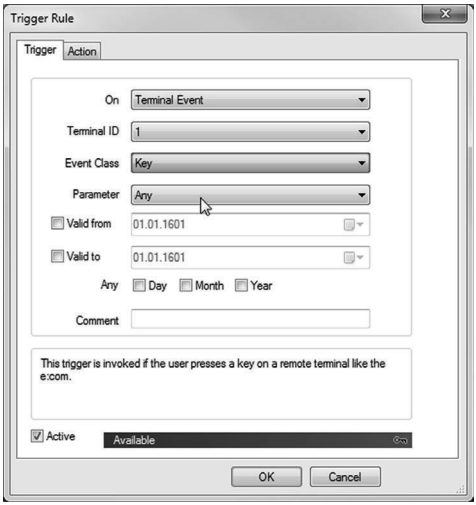
The variable device_status is a DWORD (32-bit unsigned int), acting as a bitmask. The lower 16 bit indicate which keys have been pressed. The following 8 bit (from position 17 to 24) show the status of the 8 opto inputs each e:com terminal has. Obviously, setting a bit to '1' stands for 'key pressed', while a '0' reports the opposite.

The following table shows which bit a key is assigned to.

Bit Number (Lowest to Highest):	Assigned Key:
1	Shift-Key
2	Up-Key
3	Down-Key
4	Esc-Key
5	Enter-Key
6	Sel-Key
7	F1

8	F2
9	F3
10	F4
11	F5
12	F6
13	F7
14	F8
14	F9
16	F10
17 – 24	Opto-Inputs 1 – 8
25 – 32	not used

The e:cue Programmer can react to keyboard events using Triggers. To make use of this, open the Triggers window (`View | Trigger List`, press F7 or T on your keyboard). Click the `Add` button and the following menu will appear:



There are many options to specify which Terminal and which key to react on and, of course, what to do. In this case, pressing the selected key on a terminal will let the Programmer call a macro named “KeyPressed” (Needless to say that the macro has to exist inside the show).

20.5 Network parameter set

When a greater quantity of values has to be transmitted, the system of sending keyboard events may prove to be too inflexible or complicated. For this cause the so-called Network Parameter Set exists. This is an array of 1024 variables of type `DWORD` within the Media Engine (and the e:cue Programmer respectively). You can write into each variable via UDP.

Again, the frame type `ECUE_BROADCAST` will be used. Within the packet, several `NETWORK_PARAMETER` structures can be passed into the otherwise unused field ‘extra’:

```
typedef struct
{
            DWORD parameter_id;
            DWORD value;
} NETWORK_PARAMETER;
```

A `NETWORK_PARAMETER` consists of two values: `parameter_id` names the variable number (0 to 1023) to be set in the Media Engine/the Programmer. In `value`, the new value for this variable is passed.

Up to 16 `NETWORK_PARAMETER` records can be passed inside a single `ECUE_BROADCAST` packet.

magic	MAGIC_ECUE
static_key	0
device_status	0
device_type	3 = e:com Terminal
device_version	0
event_id	3 = Network Parameter Set
event_par	quantity of Network Parameter Sets
filename	0
extra	array of up to 16 structures of type NETWORK_PARAMETER
link_status	1
mac	the sender's mac-address

20.6 Remote Command Execution and Macro Calls

As you should know, the e:cue Programmer contains its own macro language, the e:script. It is possible to remotely execute single macro commands or a complete macro via UDP. To make use of this feature, remote procedure call must be allowed within the programmer. Furthermore, the e:cue Programmer will only accept remote procedure calls of registered terminal devices. This is for security reasons because it would be very dangerous to let unknown devices use macro commands on remote machines.

We use the field `extra` to transmit the desired macro command:

magic	MAGIC_ECUE
static_key	0
device_status	0
device_type	3
device_version	0
event_id	12 = execute macro command
event_par	0
filename	0
extra	macro command
link_status	1
mac	the sender's mac-address

If we want to call a macro, again we make use of the field extra:

magic	MAGIC_ECUE
static_key	0
device_status	0
device_type	3
device_version	0
event_id	13 = call macro
event_par	0
filename	0
extra	macro name
link_status	1
mac	the sender's mac-address

Example:

To start the first cuelist via remote macro command execution, we define a new structure of type `ECUE_BROADCAST` and allocate the struct members as follows:

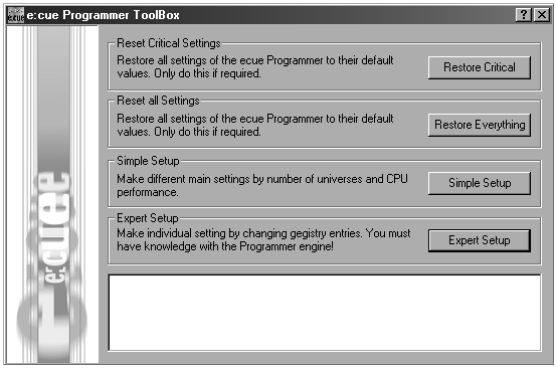
```
ECUE_BROADCAST eb;
eb.magic = MAGIC_ECUE;
eb.static_key = 0;
eb.device_status = 0;
eb.device_type = 3;
eb.device_version = 0;
eb.event_id = 12;
eb.event_par = 0;
eb.filename = "";
eb.extra = "StartCueList(0)";
eb.link_status = "1";
eb.mac = "0";
```

21 Tweaks and tricks

The following chapter includes some advanced tricks that might be useful when working with the e:cue Programmer. Most of them are pretty short and only usable in special situations. Nevertheless, you might find some nice things while grubbing through this chapter.

21.1 Programmer tuning via Toolbox

The e:cue Programmer contains a little configuration and emergency repair software utility, called Toolbox. You can start it from “Start\All Programs\e:cue Lighting Control\Application Suite 5.1\ Programmer”.



There are four different options here: First of all, if you encounter any problems using the e:cue Programmer, you can either try restoring some critical settings (Restore Critical) or if this fails restore all Programmer configuration options to their default values (Restore Everything) for troubleshooting.

Under ‘Simple Setup’ you can find a slide control to change the maximum number of universes the Programmer shall support and another one to adjust the approximate CPU speed of your machine. This will automatically tweak some internal settings to let the Programmer make use of the provided CPU power.

If you want more control over the Programmer settings, click ‘Expert Setup’. Here you can find the possibility to change several registry configuration values concerning the e:cue Programmer. You could change them directly inside the registry under

HKEY_CURRENT_USER\Software\ecue\e:cue Programmer V5.1\Options

but with the Toolbox, it is much easier. Please mind that tuning up the values can also have negative side effects on the performance. The e:cue Programmer default settings refer roughly to a PC with 2GHz CPU speed and 512MB RAM.

In the following chapter, you can find a short explanation of the particular settings.

If you experience any Problems, fall back to the default values! No warranty of any kind!

21.2 Configuration values

m_so.main_fps
Default Value: 30
Max: 35
Min: 10

This entry defines the Programmer GUI speed in frames per second.

Higher values accelerate the e:cue Programmers user interface at the cost of more CPU power. Lower values save CPU resources, giving more power to the DMX engine. If you want the e:cue Programmer to work with a very large number of universes and DMX output devices, you may want to decrease this value to save CPU resources.

m_so.sonic_fps
Default Value: 35
Max: 35'
Min: 1

This entry defines the DMX engine speed. That is how often per second new DMX frames are calculated and sent to the DMX output devices.

Lower values save CPU resources but result in a sticky DMX output.

Higher values than 35 fps are not useful as DMX is not capable of transferring more frames per seconds on all 512 channels.

m_so.universes_max
Default Value: 32
Min: 16
Max: 128

Defines the maximum number of DMX output universes.

Increasing this value costs a little of both CPU power and RAM. This applies even if there are no fixtures mapped to all universes. So you should only increase this value if you really have a demand for it. Decreasing the value below the default value of 16 DMX universes is generally NOT useful and not recommended.

m_so.devices_max
Default Value: 32 *170
Min: m_so.universes_max *170
Max: m_so.universes_max *512

This defines the maximum number of fixtures the e:cue Programmer can handle.

The default value is the number of universes multiplied by 170, as you can map no more than 170 RGB faders to a DMX universe.

For the unlikely case that all your DMX universes are fully in use and you want to use desk channels on all 512 channels the default value may be too small. Normally this never will happen. But if the case occurs, increasing the value will help.

m_so.overdrive
Default Value: 16

Do not change this Value!

m_so.handles_max
Default Value: 48
Min: 16
Max: 256

This defines the maximum number of simultaneous running cuelist. Increasing this value costs a little more CPU power and more RAM.

m_so.desk_channel_max
Default Value: 16*512

Defines the maximum number of desk channels. A desk channel is not exactly the same like a DMX channel.

However, to keep things simple, you should always use m_so.universes_max*512 here.

m_so.dmx_driver_max
Default Value: 20

Defines the maximum number of DMX output drivers, i.e. how many DMX output devices can be addressed in the e:cue Programmer device manager.

21.3 The Quick Tune up Cookbook

Parameter	Default Value	Quick Formula
m_so.main_fps	30	30 normal, 15 for very large installations to save CPU power
m_so.sonic_fps	35	35 normal, 25 to save CPU power. The latter results in a sticker, slower DMX output.
m_so.universes_max	16	16 normal 32 no problem on a standard PC (equiv. Pentium IV 3.0 GHz) 64 possible on a fast PC.
m_so.devices_max	16*170	Should be the number of universes multiplied by 170
m_so.overdrive	16	Do not change.
m_so.handles_max	36	36 or higher. Max. Number of simultaneous running cuelists
m_so.desk_channel_max	16*512	Number of universes multiplied by 512
m_so.dmx_driver_max	20	20 or higher. Depends on how many DMX output devices (e:link, e:node, butler...) you have.

21.4 Troubleshooting

If e:cue Programmer should not start up or other problems occur after tuning the values, you can do the following: close the e:cue Programmer, then press START | ecue Lighting Control | Toolbox and press the button 'restore everything' to delete all persistent e:cue Programmer settings from your Windows registry. After this, all self-made settings from e:cue Programmer / Application Options are reset to their defaults and must be reconfigured.

Tips and Tricks

The settings configured within the Toolbox expert setup are stored inside the Windows registry. You can export your current registry settings in a file by using the Windows Regedit application.

In Regedit go to

HKEY_CURRENT_USER\Software\ecue\e:cue Programmer V7.0\Options.

In the tree view, right click onto the word 'Options' and select export from the context popup menu.

Regedit creates a normal text file that can easily be adjusted with a normal Text editor like Windows Notepad. (Do not use Word or WordPad for this job!)

You can reimport the settings from your exported file by simply double clicking on the file. Windows will ask you whether you want to reimport the settings to your registry again.

Please mind: Wrong usage of the Windows Regedit tool can damage your Programmer configuration and even your complete Windows installation! No warranty of any Kind!

21.5 Tweaking your workflow

Programmer View and cues

- As long as you hold down the SHIFT key, the logbook will not be updated. If there is much output in a logbook window, you can suppress updates for a while this way. This simplifies reading the text.
- You can merge multiple cues in the Programmer View like this: Open the cuelist window, select a cue and press the LOAD icon. Select another cue, hold the SHIFT key and press the LOAD icon again. Now the content of both cues is merged in the Programmer View and can be saved as a new cue.
- You can select multiple groups in the Programmer View by holding the CONTROL key when selecting the groups with the mouse.
- You can move presets in the preset window by selecting them and then pressing CONTROL and the cursor keys.
- To modify channel values more slowly, hold down the shift key when moving with the mouse. (At "Application Options / Mouse", you can define the mouse sensitivity).
- To spread channel values select some fixtures and hold the CONTROL key when modifying the channel's values.
- A channel value in the Programmer View, which is loaded with a preset, will not change when turning any of the wheels, except when you hold the ALT key on the keyboard. This re-enables modification of the channel value.

21.6 Macro text editor

- You can indent your current text selection by pressing TAB.
- You can remove indentation of your current text selection by pressing SHIFT+TAB.

21.7 Using the Video Wizard on multiple sections

To use the e:cue Programmers Video Wizard on multiple Sections, adhere to the following steps:

- Call the Video Wizard.
- Select the fixtures of section 1.
- For the horizontal direction, enter Min = 0 % und Max = 33 %.
- On the tab ‘Action’, set back the action type to ‘--’ so that no action will be executed.
- Press OK. The first cue can now be found inside the cuelist.
- Call the Video Wizard a second time.
- Select the fixtures of section 2.
- For the horizontal direction, enter Min = 34 % und Max = 66 %.
- On the tab ‘Action’, again set back the action type to ‘--’.
- Press OK. The second cue can now be found inside the cuelist.
- Call the Video Wizard a third time.
- Select the fixtures of section 3.
- For the horizontal direction, enter Min = 67 % und Max = 100 %.
- On the tab ‘Action’, again set back the action type to ‘--’.
- Press OK. The second cue can now be found inside the cuelist.
- Highlight the first cue of the cuelist and load it into the Programmer View by pressing the ‘load’ icon.
- Highlight the second cue of the cuelist, hold the shift key and load it additionally into the Programmer View by pressing the ‘load’ icon.
- Highlight the third cue of the cuelist, hold the shift key and load it additionally into the Programmer View by pressing the ‘load’ icon.
- Save the jointed cue as a new cue into the cuelist by pressing ‘record’.
- Edit the cue (double-click the cue). Insert an appropriate Media Play action.
- Optional: Insert a Media Stop action into the cuelist as a release action.
- After approximately 30 minutes the cake is finished and can, after cooling down a short time, be served with whipped cream.

22 Notes

e:cue

LIGHTING CONTROL

Downloads and more information at www.traxontechnologies.com and www.ecue.com