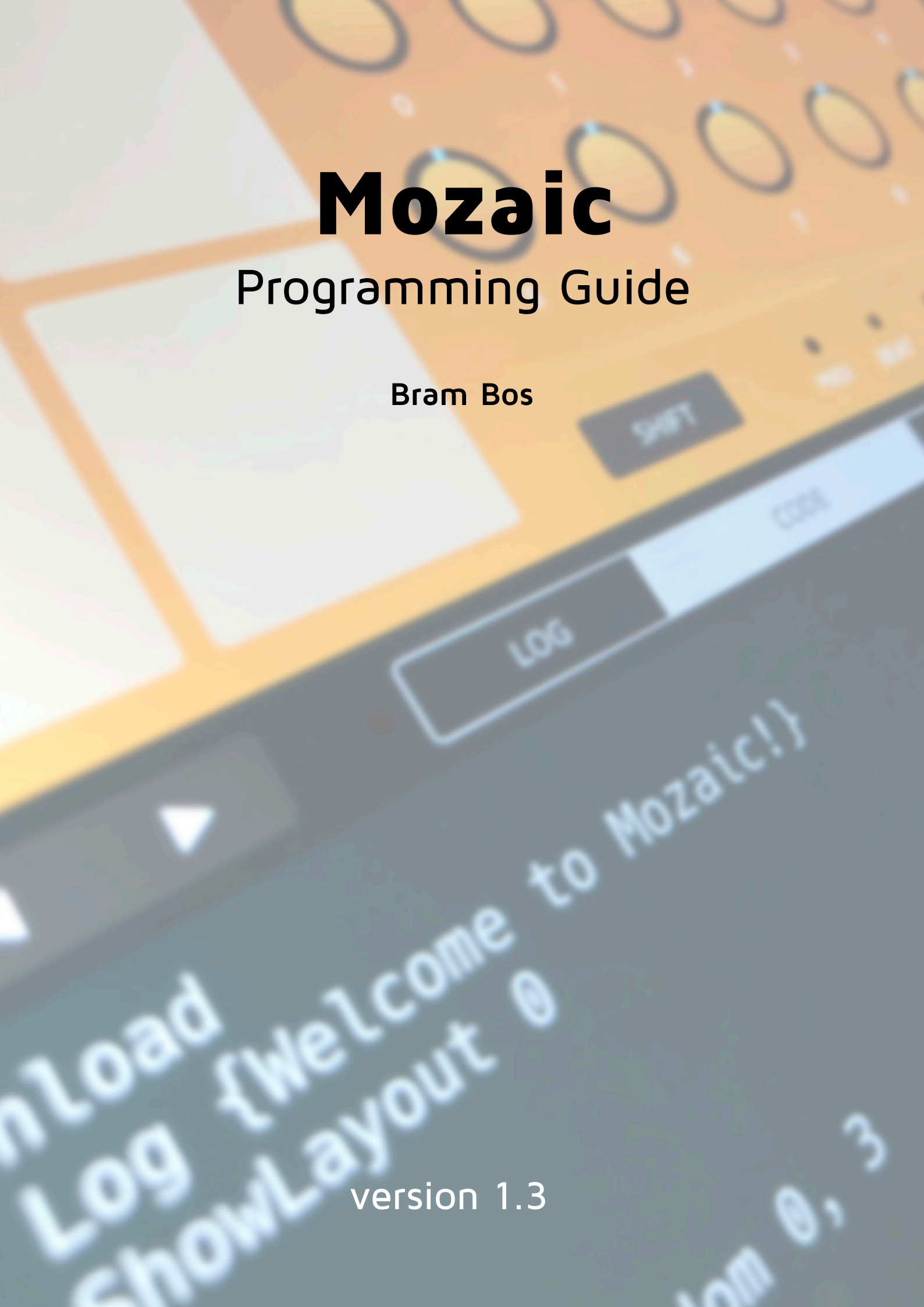


Mozaic

Programming Guide

Bram Bos

version 1.3

Table of Contents

0. Introduction

So you have a great idea for a MIDI effect plugin? Or your amazing MIDI setup is misbehaving because you can't get the MIDI channels to play nice between apps and input devices? Is your concept is too niche or is your use case too specific for any of the mainstream apps to provide a solution?

Why not roll your own?

Creating an Audio Unit plugin on iOS is not for the faint hearted. Apple's MIDI plugin standard is notoriously un[der]documented and getting everything to work on available hosts can be an arduous process. And then we're not even mentioning the process of getting the plugin officially approved and into the App Store. Mozaic is here to help you out.

Mozaic is centered around a simple, yet powerful script language - specifically designed to offer MIDI manipulation and generation with the least possible effort. Built-in functionality for LFOs, melodic scales, host-plugin-communication and ready-to-use GUI layouts let you set up snazzy plugins with just a few lines of code. You even get access to the tilt sensors of your device using just a single function call!

To help you get started, the plugin comes with a lot of examples and tutorial content. Additionally, I've written this Programming Guide to talk you through the basics of the Mozaic script language and its treasure chest of powerful functions you can tap into.

The language and the integrated editor have been designed to be fun and accessible, and to immediately reward any effort you put in.

Sit down with an iOS device, this guide, a hot beverage of choice, and prepare to impress yourself.

1. Mozaic Script 101

Intro

The full power of Mozaic is unlocked through its Mozaic Script language. The thought of learning a programming language may seem daunting, but unlike regular 'general purpose' languages such as C++ and Swift, Mozaic Script was made specifically for MIDI applications. This means all the complex heavy lifting is done for you, and you can e.g. set up and use an LFO with just a single line of code.

So is Mozaic Script much like C++ or JavaScript? Not exactly. For starters, it borrows quite a few design principles from Pascal: a language which was designed with an emphasis on human readability. As a result it is a bit more ‘wordy’ than C-inspired languages, making it easier for others (or for your future self) to look at your code and understand immediately what's happening. In short, Mozaic Script was designed to resemble written language so it can be learned and understood by looking at examples.

C:

```
if ( counter != 64 && hostBeat() == 0 ) {
  // do something
}
```

Mozaic Script:

```
if counter <> 64 and HostBeat = 0
  // do something
endif
```

As you can see, Mozaic Script uses fewer special characters and more "human" language elements (e.g. "and" instead of "&&"). Also, Mozaic Script is not case sensitive, so mixing up lower case and upper case is not going to result in errors. It's much more forgiving that way.

Mozaic Script is not object-oriented (OO), which makes it more beginner-friendly than full-blown OO languages. Obviously, the technology behind the Mozaic AUv3 plugin is deeply object oriented, but all this is kept far away from the user.

Libraries full of books have been written about computer science, and programming theory in particular. I will touch on a few key concepts in this document, but if you like to get a proper coding background there are plenty of other sources of wisdom, both on paper and online. In the next few paragraphs I will explain some programming basics which are standard fare in many contemporary languages. So whatever you learn here will also come in handy should you ever want to move on to full-blown programming environments.

The essence of programming

When you're writing a program, you are tackling a seemingly big, complex task by dividing it into simple little sub tasks. For example: a task like "make a fire" can be split up into smaller steps:

- find a good place to make the fire
- find wood
- check if wood is dry enough, if not: find more dry wood
- get a box of matches
- etc.

Code is not that different: it's merely a sequence of simple instructions you're telling the machine to perform, which together make up a bigger job. In pseudo-code (i.e. fake code-like words, laid out to explain a concept) that could look like:

```
Start
Do something
Do something else
Do yet another thing
Done
```

In this example the computer (or in our case: our iOS device) is always doing the same thing. No, that's not very interesting.

That's why there are certain logical structures you can apply to your code, in order to make your script do useful things at the right time. Mozaic Script offers 5 distinct mechanisms to help you build a meaningful application. When you understand these 5 principles, you know the fundaments of (almost) everything there is to know. All the rest is just a matter of practice :-)

We'll go into details for each of these mechanisms, but let's quickly introduce them here, using our pseudo-code example.

Events
Events help you by automatically running parts of your script when they are needed. For example, when the user sends MIDI into your plugin, you can tell the script to start a snippet of code you wrote specifically for handling MIDI input:

```
When MIDI input is detected: Start
Do something
Do something else
Do yet another thing
Done
```

Conditionals
These check if a certain condition is met, and execute (or skip) certain parts of your script accordingly. For example: if the sustain pedal is currently active, you want the code to be executed, or do some other things otherwise:

```
When MIDI input is detected: Start
Is the sustain pedal active? If so...
   Do something
Is the sustain pedal not active?
   Do another thing
Done
```

Loops and iterations
These are 3 different mechanisms for repeatedly executing the same code snippet. One repeats a set number of times, and the others repeat if -or until- a certain condition is met.

```
When MIDI input is detected: Start
Repeat the following instructions...
   generate a random value between 0 and 5
   write the value on the screen
...until the value is 0
Done
```

or:

```
When MIDI input is detected: Start
Set a value to 0
While the value is less than 10,loop the following instructions...
     write the value on the screen
     add 1 to the value
Done
```

or:

```
When MIDI input is detected: Start
Repeat the following instructions 50 times:
   generate a random value
   write the value on the screen
Done
```

That's all there is to it! If these mechanisms make sense to you (even in fake code) you know how to think like a computer programmer. Now let's see in a bit more detail how we can write such things (and slightly more useful things) in actual working script...

Events

Mozaic Script is an "event based" language. Rather than being a linear program which runs from start to end, a Mozaic program is a collection of snippets, which describe what the program will do when certain things happen. E.g. you can have event handlers which describe what actions to take when...
- an incoming MIDI event is detected
- the AU host starts playback
- the user changes a knob on the screen
- the user presses/releases the sustain pedal
- the host's timeline reaches the next beat or the next bar

The Mozaic plugin automatically tracks all these events for you; but only when you assign a code snippet to such an event something will happen. All the difficult AUv3, MIDI and GUI stuff is handled in the background. The script just needs to specify what needs to happen and doesn't need to worry about managing Apple's labyrinthine system frameworks.

For example, if I want a MIDI note to be sent on every beat, you only need to add this snippet to the code:

```
// send note #35 to channel 0, with a velocity of 100
// queue a note off for note #35, 300 milliseconds later

@OnNewBeat
   SendMIDINoteOn 0, 35, 100
   SendMIDINoteOff 0, 35, 0, 300
@End
```

That's all. *@OnNewBeat* is one of several events provided by Mozaic. Behind the scenes the plugin keeps track of the host's timeline and transport state for us. Whenever the next beat is reached this script is automatically fired up and all the MIDI communication is handled for us. Sample accurate timing, transport state awareness, host synchronization, AU MIDI communication, event queuing. All with just 4 lines of code - instead of thousands (literally).

Also note the "remarks". Everything typed after "//" is considered a remark and will be ignored by the script. You can use remarks to guide people who read your code through whatever it is you're doing and explain your thoughts. Remarks don't do anything functionally, but are incredibly important if you're sharing your scripts with others, or just want to be able to read your own code later on.

Functions and commands

Functions are the meat on the bones of your application; they actually perform all the magic tricks. Mozaic Script comes with a sizeable library of functions and commands tailored to doing MIDI and music related operations. In the previous code snippet, SendMIDINoteOn and SendMIDINoteOff are examples of commands. You call them and give them a set of parameter values and then they're off doing interesting stuff for you.

Functions do something similar, but also give you something back. In a way functions are comparable to modules in a modular synth. Each module is a little black box which can have inputs, outputs and parameters. Let's spruce up our previous code and add a function to randomize the note we're playing on each beat:

```
@OnNewBeat
   note = Random 0, 127
   SendMIDINoteOn 0, note, 100
   SendMIDINoteOff 0, note, 0, 500.0
@End
```

The new line is of interest here. We call the "*Random*" function with two parameters: 0 and 127. This means we want the script to generate a random number in the range of 0-127. The line begins with "*note* = ". We'll discuss variables next, but basically this bit means: we'll put the output of the random function in a little box, and we're labelling this box "*note*" for later use.

In the next two command calls, "*SendMIDINoteOn*" and "*SendMIDINoteOff*", we're getting the random value out of its box and use it instead of the fixed value of 35 from our last example. So instead of always using the same value, we're using whatever was put into "note" - which happens to be our random value in this example code. We just made a script that plays a random note every beat!

In Mozaic Script you can write your own ‘User Events’, which can be used to create your own functions. These will be discussed in one of the following chapters.

A complete alphabetical list and documentation of the available functions and commands is provided in Chapter 7 this document (and Chapter 8 has an overview grouped by type of functionality).

If you’re worried about having to remember all the parameters needed for each function; Mozaic will help you quite a bit. There’s code-completion in the IDE, and you always have access to the Syntax Lookup window which shows the exact sequence of all functions available in Mozaic!

Variables

Variables are cool. They are little boxes of memory which let us store values for later use. A variable has a name - which you can make up - which lets us reference it and manipulate it using commands and functions and other script statements. Some examples:

```
velocity = Random 20, 100 // put a random value between 20-100 into ‘velocity’
a = 35.5 // put a fixed value of 35.5 into ‘a’
sum = v1 + v2 // put the sum of v1 and v2 into sum
newcommand = 0x90 + MIDIChannel // add MIDIChannel to hexadecimal 0x90
```

Creating variables
Variables are automatically managed for you as soon as you create them. You create a variable simply by assigning a value to it:

```
newvariable = 0 // from now on you can use ‘newvariable’ anywhere in your script
```

If you try to read or otherwise reference a variable before it has been assigned an initial value you will get a syntax error in your Log window. You can’t use a variable before it exists as it won’t have any valid or meaningful value.

It is good practice to create and initialize variables you want to use across events inside the *@OnLoad* handler, because that ensures they will exist whenever you need them.

Variable types
In other languages there are many different types of numerical variables; integers, bytes, floats, doubles, booleans, etc. Mozaic Script variables are smart: you don't have to worry about variable types and you can use any variable for any purpose. The Mozaic engine will figure out what datatype is most appropriate for each situation and morph your values accordingly while retaining maximum accuracy. It's actually a pretty nifty system, but the main benefit is that you don't have to worry - or even know - about it.

Scope
Variables in Mozaic Script have an instance-wide "scope". This means that their contents can be retrieved from any event call at any moment after you set them (and remain valid until you change them) within the current plugin instance.

So you can, for example, set up some default values for your variables in the *@OnLoad* event (which is called as soon as you load or update your script). These values will then be retained and be accessible from all subsequent calls to any event handler.

Naming
You are free to come up with descriptive (or completely puzzling) names for your values. Names can be a single character long (‘x’) or hundreds of characters if you want. However, there are some rules and guidelines you should stick to:

Only use alphanumeric characters; do not use special characters (with the sole exception of underscores) or spaces in names!

```
myVariable = 1
x = 0.5
box1 = box2 + ball3
nameWithoutSpaces = 0
name_without_spaces = 1000.0
```

Names are case-insensitive, so ‘aaa’ is the same as ‘AAA’ or ‘aAa’.

```
myvar = 5
Log MYVAR, mYvAr //this will log ‘5 5’
```

Obviously you should not use ‘reserved words’; words already used for Mozaic functions and commands. Also don’t start a name with ‘MIDI’ as this may conflict with (future) built-in functions and variables.

```
firstbyte = MIDIByte1
secondsbyte = MIDIByte2
rand = Random 0, 100
```

Use descriptive names you would understand if you read back the script a year later.

```
noteCounter = noteCounter + 1
velocityPercentage = 100
recordBuffer[bufferPointer] = 0
```

Variables have an optimal accuracy of up to 24 bits. Numbers which exceed the 24 bit range will be captured, but may lose accuracy because Mozaic will start treating them as 32 bit float values. This should be plenty of headroom for typical MIDI use cases however.

Global ‘Meta Variables’

Mozaic also offers 100 meta-variables. These are variables which exist across all active Mozaic plugin instances. They are always named GLOBAL0 - GLOBAL99. Use these meta variables with a lot of caution. The validity of their contents can not be guaranteed as any Mozaic plugin has access to them and can alter their values at any time. Meta variables always exist and don’t need to be explicitly created. You access them like this:

```
GLOBAL0 = MIDIByte2
myLocalVariable = GLOBAL2 + GLOBAL3 + GLOBAL4
```

Meta variables make use of an undocumented side-effect of how AUv3 plugins work inside iOS ‘sandboxes’. Although unlikely, Apple may decide to change this behavior at some point.

Arrays
If you were wondering about "arrays": every Mozaic variable can be used as an array of up to 1024 cells (arrays are automatically generated for you when you need them, no need to declare or reserve anything).

Arrays are sets of variables (or cells) packages in a single variable. It's like a box full of little compartments. You can access these cell values in the usual way:

```
fractions[0] = 12.5 // put 12.5 in the 1st cell of the array named fractions
v[64] = 0           // set the value of cell 64 of 'v' to 0
record[pointer] = 0 // set value of the cell pointed to by 'pointer' to 0
v[999] = 0xFF       // put hexadecimal value FF into cell 1000 of var 'v'
```

Note that the first cell in the array is the same as the base variable:

```
// the regular variable is always the same as the first cell (0) in its array
xyz[0] = 3          // this is the same thing as the following...
xyz = 3
```

Cells in an array can be initialized 'in bulk' for convenience:

```
ruismaker_notes = [49, 51, 54, 56, 58, 61, 63, 66]
```

This fills the first 8 cells in *ruismaker_notes* (cells 0-7) with the listed numbers. This is much more convenient than assigning all 8 cells in separate script lines. You can even use array indices in these initializers to specify at which point in the array the values should be stored…

```
buffer[100] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
states[5] = [YES, NO, YES, NO, NO]
```

Copying arrays
Arrays can also be (partially) copied, using the *CopyArray* function:

```
CopyArray src, dst
```

This copies the entire array *src* into array *dst*. However, for performance reasons it is recommended to specify how many cells you want to copy (if you know you only need a limited number).

```
CopyArray foo, bar, 10
CopyArray foo, bar[100], 10
CopyArray foo[100], bar[100], 10
```

In all three examples above only 10 cells are copied, which is much faster than copying all 1024 cells the full array can contain. The difference between these examples is which 10 cells are being copied and at which point in the destination array they will end up.

Advanced shizzle: variables, initialization and state-saving

The contents of your variables are always automatically saved as part of the Audio Unit state saving process. So their values will be remembered when you save them as a preset, or when you use the plugin in a host and save a host project (which will capture the latest state of your script and store it).

Say you've written a MIDI looper script. When the user runs the script, records MIDI into your looper and then saves the plugin's state, it will also save any recorded MIDI data and other settings you're keeping in your variables for next time. That's neat!

But as we've mentioned before, it is good practice to initialize variables in the *@OnLoad* event. However, that would erase any variable you just loaded from the file, preset or host project. Hmmm… That's an unwanted side-effect. If only there was a way of knowing whether we're running the script for the very first time (in which case we want to initialize) or just loaded a previously saved preset from a file (in which case we don't)… *There is!*

```
@OnLoad
  if Unassigned testvariable
    // testvariable does not exist yet,
    // so we know we're running the script for the first time!
    // we can safely create and initialize it!
    testvariable = 0
  endif
@End
```

We use the function "*Unassigned*" to check if a variable already exists and contains a valid value. If *Unassigned* returns *true* or *YES*, this means the variable we're checking does not exist, and we can safely proceed to create and initialize it without worrying we're overwriting any existing value!

It's usually not necessary to test each variable independently. If one exists, the others likely do too. So just test for one key variable and if you need to initialize that one, you can safely initialize all the others in the same go. Also, if you don't care about previously saved values, you don't need to check at all.

```
@OnLoad
  if Unassigned recordbuffer
    // create and initialize the record buffer and record pointer
    FillArray recordbuffer, 0
    recordpointer = 0
  endif

  // this variable will always be initialized:
  xyz = -1
@End
```

Note: testing a variable with Unassigned does not automatically create the variable. To create the variable, you still need to assign a value to it. After you've done so, the variable will no longer be unassigned.

Conditional Processes

One of the most important aspects of programming is the use of conditional process flows. Very simply put, these are a way of specifying *"if this condition is true, do the following. Otherwise, do this other thing."* In most programming languages, this is done through "if"-statements. Mozaic Scripts are no different.

Look at the following example:

```
if HostRunning
  Log HostBar, HostBeat // print the current bar and the current beat
else
  Log {Host is not playing!}
endif
```

This bit of code will, when placed in an event code block, check the current state of the host. If the host transport is running it will display the current bar and beat in the console. However, if the host is not running it will execute the line following the "else" statement instead.

This is a simple example, but you can probably already sense that if-statements are an extremely powerful programming pattern, letting you embed lots of clever smartness in your scripts. Each level in a conditional statement is closed with an "endif" statement. This becomes very important when you start nesting conditional statements.

Sometimes you want to test for several different conditions. In this case we can extend the "else" statement into an "elseif" statement:

```
if velocity = 64
   // do something when velocity is exactly 64
elseif velocity > 64
   // do something else when velocity is higher than 64
else
   // velocity is smaller than 64
endif
```

We can also use nested "if" statements. Nested conditionals means that you put conditional statements inside conditional statements (inside conditional statements, etc.).

```
if v1 = 0
   if v2 = 0
      // do this if both v1 and v2 are 0
   endif
endif
```

Nice! But there's an even more elegant way to do this...

We can rewrite our nested if-statement as a so-called *compound* if-statement. Compound conditionals use "and" and "or" operators to create a list of conditions we want to test for in one go. The above example rewritten with a compound structure instead of a nested structure would look like this:

```
if v1 = 0 and v2 = 0
 // do this if both v1 and v2 are 0
endif
```

You can use as many levels of nesting as you need (and really long sequences of compound statements). But if you need excessive levels you may want to rethink your code structure because there are probably more efficient ways to do what you want to do.

Complex nested compound statements are allowed in Mozaic Script. Consider the following situation:

```
if v1 < 64
   if v2 < 20 or v2 > 100
      // do something useful
   endif
endif
```

You can also rewrite this using compound logical conditions (with "and" and "or" statements), where conditions in brackets are evaluated separately:

```
if ( v1 < 64 and ( v2 > 100 or v2 < 20 ) )
   // that's much more efficient!
endif
```

'Boolean' states
When you do a conditional test, you are actually checking a boolean state. That's just jargon for something which is either *true* or *false*. You can use any variable for that. If you don't explicitly state whether you're looking for true or false, true will be assumed, leading to code which is nicer to read:

```
cool = YES

if cool
  Log { That's cool! }
endif
```

These are the boolean states you can use in your scripts:

```
True, YES or 1 (or any other non-zero value)
False, NO or 0 (zero always means false)
```

For the math-boffins

Maths may not be everyone's favorite subject, but you'll be happy to know that you can insert mathematical operators (almost) anywhere in your scripts, and that Mozaic respects the mathematical rules of precedence.

```
xyz = ( HostBar * 4 / 1.5 ) + 3

SendMIDINoteOn 0, MIDINote + 12, MIDIVelocity - 20

difference = Abs ( var1 - var2 ) + Abs ( var3 - var4 )

ugh = ( ( HostBar * HostBeatsPerMeasure ) + HostBeat ) % 4

latchState = NOT latchState
```

Mozaic supports the following operators:

Basic mathematical operators	`a + b - c * d / e`
Modulo	`a % b`
Integer And	`a & b`
Integer Or	`a | b`
Integer Xor	`a ^ b`
Unary - (i.e. negate)	`-a`
Boolean NOT	`NOT a`

Logical operators	`a = b`
	`a < b`
	`a > b`
	`a <= b`
	`a >= b`
	`a <> b`
	`a AND b`
	`a OR b`

Boolean states	`YES, true, 1 (or any non-zero value)` `NO, false, 0`

For a full overview of all built-in math functions refer to the overviews in chapters 9 and 10.

Loops

Sometimes you want the same bit of code to be executed repeatedly. This can be done using loops. There are 3 types of loops available in Mozaic Script: “for”-loops, “while”-loops and “repeat until”-loops.

Repeat
Starting with the latter, you can use "repeat"-statements to loop a piece of code for an unspecified number of times until a certain condition is met. So it's a combination of a loop function and an if-statement, like we've already discussed. Look at this code snippet:

```
repeat
   rnd = Random 0, 9    // this bit will be executed repeatedly...
until rnd = 0           // ...until the random number 0 is generated

// Note: in theory this could be an endless loop,
// so this is not good programming practice ;-)
```

The conditions you put after the “until” statement follow the same rules as the conditions you put in if-statements (i.e. they can also be compound, use brackets, etc.). So these should be familiar to you by now.

Note that plugin and system states are not updated while you are inside a loop. So if you write the following example, you may end up in an endless loop because HostRunning will not be updated when you’re already inside this event handler.

```
// don’t do this!!!
repeat
   Log {This may loop forever!}
until HostRunning = NO
```

While
Note that a repeat-until loop is always executed at least once, because the condition is tested at the end of the loop. In some cases you have code that you want to loop, but only after you have tested for a condition up front. You could wrap a repeat-until loop inside an if-statement to achieve this, but Mozaic script has a different loop method for you: the while-loop.

Check this example:

```
rnd = Random 0, 3
while rnd > 0
    Log rnd
    rnd = rnd - 1
endwhile
```

What do you think happens here? First, a random value between 0 and 3 is put into ‘rnd’. Next, we enter the while-loop, but only if rnd is greater than 0. If the Random function generated a value of 0, the entire

while-loop will be skipped. As you have probably guessed: the while loop starts with the “while” statement, and ends with the “endwhile” statement. Inside the while-loop the value of rnd is printed on the Log-screen, and we subtract 1 from rnd. This bit is repeated until rnd has become 0.

So the main difference between *repeat* and *while* is that the while-condition is checked before entering the loop, and the repeat-condition is checked when ending the loop. So the repeat-until loop is always executed at least once, whereas the while-loop can theoretically be skipped entirely.

For
"For"-loops are a clever little programming pattern which is typically used for cases when you know how often the loop needs to be repeated. It doesn't use a conditional statement but requires you to tell the code how many iterations you want:

```
for count = 0 to 15
   Log count
endfor
```

This will loop 16 times. It starts by putting value 0 into count. Then it will go through the code between for... and ...endfor and increase the value of count each loop. So in this example count is used as a counter, starting at 0 and ending after count has reached 15 (so: 16 iterations in total). Note that you can count down using the same notation:

```
for countdown = 50 to 25
  Log countdown // we're counting down now!
endfor
```

You can also parametrize this for-loop, which makes it a clever and useful mechanism:

```
times = Random 1, 5
for index = 0 to times
   Log times, index
endfor
```

Or do something like this:

```
for knob = 0 to 21
   LabelKnob knob, {Knob }, knob
endfor
```

User Events (Subroutines)

We've already seen Mozaic events, such as *@OnLoad*, *@OnMidiInput*, *@OnNewBeat*. As of version 1.0.2, Mozaic lets you define your own events (or subroutines, or procedures, or whatever you want to call them). Your User Events will never be automatically called by Mozaic, but you can call them from your script whenever you need.

Making your code more efficient
Imagine you have a piece of code that appears multiple times in your code (such as updating the label of a knob). You could put that bit of code in a *user event* and call it every time you need it. Not only helps this keep your script shorter, but it also makes sure that if you're fixing bugs or updating code, you only need to do it in one place!

```
@OnLoad
  knobnum = 0
  knobvalue = 64
  Call @MyKnobLabel
@End

@OnKnobChange
  knobnum = LastKnob
  knobvalue = GetKnobValue LastKnob
  Call @MyKnobLabel
@End

// this is the new user event:
@MyKnobLabel
  LabelKnob knobnum, knobvalue
  SetKnobValue knobnum, knobvalue
@End
```

In the example above, there we've defined one user event, called *@MyKnobLabel*. This event updates the label of a knob and sets its value. The event uses two variables (*knobnum* and *knobvalue*) to determine which knob to update and what value to set it to. So we have to make sure these values are set before calling the event. We then use the *Call* command to invoke our event.

After *@MyKnobLabel* is finished, we return to where we came from and continue from there. Obviously you can also call other events from your user events.

Note: many languages let you pass 'parameters' to functions. Mozaic doesn't do that. All variables in Mozaic have a global scope. Passing function parameters would introduce variables with a local/limited scope which would come with a new level of abstraction and complexity. The absence of function parameters is a small price we pay in order to keep Mozaic simple and accessible for everyone.

Recursion
If you enjoy living on the edge, you can even call a user event from within itself. This is called recursion, and it's incredibly risky because it's all too easy to create an infinite loop and hang your plugin. So you always have to make sure that there's a solid exit strategy built into a recursive script.

However, in some exotic cases they can be really powerful. For example, the most common algorithm for calculating Euclidean divisions is based on recursive logic. As a little exercise, have a look at the following code and try to predict what it does:

```
@OnLoad
  a = 0
  Call @MyRecursiveScript
@End

@MyRecursiveScript
  if a < 100
    a = a + (Random 10, 25)
    Log a
    Call @MyRecursiveScript
  endif
@End
```

If you try it out, you'll see what it does: it's using the recursive event to keep adding a random value to *a*, until the 100-barrier is crossed, and then it safely exits. Without the *if/endif* check the script would get into an endless loop and you would probably need to close the plugin from the host (if it doesn't crash eventually due to running out of resources). That's not what you want to happen while recording or performing, so use recursion with care!

Naming User Events
You can give your user events any name you like, as long as it starts with an @-sign, just like Mozaic's own system events. I personally start all my own user events with *@My...* because it offers a nice visual distinction between system event handlers and my own custom events; but this is certainly not mandatory. Just like the rest of Mozaic's syntax, User Event names are not case-sensitive.

```
@OnLoad
  Call @FOO
  Call @BAR
@End

@Foo
  // this is ok
@End

@Bar
  // this is also ok
@End
```

Some script writing rules

Compared to other languages Mozaic is pretty liberal in its writing rules. It doesn't make sense to use a very strict language on a platform where typing requires a bit more effort. So some extra thought was put into making Mozaic Script efficient and forgiving. Yet there are some rules you need to follow.

1. One instruction per line
In your scripts, each instruction is put in its own line. Unlike other languages you can't simply sprinkle some semicolons here and there and cram 20 functions in a single line.

2. You can embed function calls as parameters for other functions
This will greatly enhance scripting efficiency. For example:

```
v = GetKnobValue 5
r = Random 0, v
l = LastKnob
SetKnobValue l, r
```

This can be rewritten more briefly as:

```
SetKnobValue LastKnob, (Random 0, (GetKnobValue 5))
```

The important rule here is that if your embedded function requires any parameters (e.g. Random requires two) you'll have to put the entire call between parenthesis. If the call does not require any parameters you can simply put the function name there (e.g. LastKnob requires no parameters of its own hence it's not put in parenthesis).

3. When in doubt, use parentheses
In some cases parentheses are mandatory, but you can use them anywhere you like to make your scripts better readable or to avoid ambiguities. For example:

```
if x < 4 and y > 10 or z < 4
  Log {something}
endif
```

Although this will work (all logical operators are evaluated from left to right) the writer's intentions are highly ambiguous. This should probably be written as:

```
if (x < 4 and y > 10) or (z < 4)
  Log {something}
endif
```

Also for readability, using parentheses often helps:

```
SetKnobValue LastKnob + 1, value * 2 - 1     // this works ok
SetKnobValue (LastKnob + 1), (value * 2 - 1) // but this is clearer!
```

2. From noffin to boffin in 60 minutes

So you're eager now to start writing Mozaic scripts? The following chapter is a set of beginner lessons to get you started with the basics of scripting, and using Mozaic in particular. Lots of tutorials can also be found in the presets list of the plugin for you to try out and tinker with.

Lesson 1: Hello World

Hello World is the classic first program anyone writes when diving into a new programming language. So to start our Mozaic journey properly we'll write a script which writes the legendary words *"Hello World!"* to the Log window when loading the script.

Open the script-window in the plugin and remove the boilerplate template code that is prefilled for you. Replace it with the following lines:

```
@OnLoad
   Log {Hello World!}
@End
```

Now run the script (press the green “Upload” button). The automatically switches to the Log window. If you did everything correctly you should see our message listed there. Let's go through the lines one-by-one.

`@OnLoad`

This indicates that the next code section will be triggered on the Load event. Essentially, it means this bit of code will be executed as soon as the script is run or the preset is loaded. This makes the OnLoad event a great place to perform initialization tasks, such as assigning default values to variables you're planning to use.

You can not have multiple event handlers for the same event; so there can only be one OnLoad handler in each script. But ofcourse you can have handlers for different events in the same script.

`Log {Hello World!}`

The Log command writes contents to the Log window. This can be a list of variables, constants and/or text strings. In this case we only write a single fixed text string. Text strings are contained within accolade brackets: *{your text here}*

`@End`

This is the standard closing line for any event code section. Other event handlers can follow after this line, but will not be executed until their respective events are triggered. Every event handler must end with this line, so the script knows where to stop executing code for this event.

Congratulations! You just officially wrote your first working Mozaic script.

Lesson 2: Responding to MIDI input

Now let's move on to a script that is somewhat more interesting. Very often you'll want to write a script that processes MIDI for you.

Mozaic will notify you when MIDI input is received. But it will not automatically forward the MIDI to its output. So if you want to send MIDI THRU, you will need to add that into your script:

```
@OnMidiInput
   SendMIDIThru
@End
```

If you put this code in your script, it will trigger the OnMidiInput event whenever MIDI comes in, and immediately forward everything to the output.

Still not particularly interesting. So lets rewrite it so it will send all incoming MIDI through, but reroute everything to one specific MIDI channel. Since this is a use case that's relatively common, Mozaic has a convenient command for that specific task: *SendMIDIThruOnCh*

```
@OnMidiInput
   SendMIDIThruOnCh 9
@End
```

This script will catch all MIDI we send into it, and forward it to channel 9 (which is the 10th channel, because we start counting at 0) - which happens to be the drum channel in many setups.

To make it even more interesting, let's filter out any MIDI CC messages. This is probably a good moment to Google a list of MIDI commands for reference. Such a list will tell you that the MIDI command for MIDI CC messages is B0 (where the '0' refers to channel number 0).

Some important MIDI Commands… (full list here: https://www.midi.org)

	Note On	Note Off	CC	Program Change	Aftertouch	Pitchbend
Command (hex#)	0x90	0x80	0xB0	0xC0	0xD0	0xE0

0xB0 is a hexadecimal number, which translates to 176 in normal human decimal numerals (you can use either in Mozaic, so choose whatever you're most comfortable with).

> Tip: Mozaic supports both decimal and hexadecimal numbers. For hex numbers the common C notation is used; just add a "0x"-prefix before the hex number: 176 = 0xB0

So we've established that the standard MIDI CC command is 0xB0. Let's add a check for that message in our script and rewrite it so that it only sends MIDI THRU when the incoming event is *not* a CC command.

```
@OnMidiInput
   if MIDICommand = 0xB0 // B0 is the standard MIDI CC event
      Log {we've caught a CC event!}
   else
      SendMIDIThruOnCh 9
   endif
@End
```

In the script we've added an if-statement around the *SendMIDIThruOnCh* command. In the if-statement, we're checking the value of "*MIDICommand*" which contains the number of the MIDI command of the even we're currently handling.

To clarify: when *@OnMidiInput* is triggered, *MIDICommand* will contain the MIDI event that was last received. To make things simpler for us, the channel number of the command will be set to 0 (we can retrieve the original channel using *MIDIChannel* value). This way we only need to do a single check (look for 0xB0) instead of 16 (look for 0xB0 - 0xBF). In short: we can disregard different MIDI channels here and just look for 0xB0.

Try modifying the code example to filter out Note On and Note Off messages.

Lesson 3: Using a Knob to send out CC

Mozaic offers a number of different layouts in its GUI which you can use any way you like. Depending on the chosen layout, these are full of pads, sliders and knobs. We will now program one of those knobs to send out MIDI Volume commands when you tweak it.

Regardless of which layout we choose, the first knob on it will be number 0. This is the one we're going to use; so we'll label it "Volume".

The other thing we need to know is that in standard MIDI, changing volume is done by sending out MIDI CC 7, with the desired volume level as its value. MIDI CC values are always in the range 0-127 (i.e. 128 discrete values, which are then smoothed and interpolated by the receiving instrument).

Here's the full script. Run it and try to figure out what's happening:

```
@OnLoad
   ShowLayout 1
   LabelKnob 0, {Volume}
@End

@OnKnobChange
   if LastKnob = 0
      setting = GetKnobValue 0
      SendMIDICC 0, 7, setting // send out CC#7 on channel 0
   endif
@End
```

The first interesting observation is that there are two event handlers in the script. The *OnLoad* event is triggered when you press the upload button in the code-window, when you load the script as a preset, or when you open a project containing the Mozaic plugin.

In the *OnLoad* event we select layout 1 (the second layout, because we always start counting at 0). Next we change the label of the first knob (number 0) into "Volume". That's nice and descriptive.

The *OnKnobChange* event is new for us. As you can guess, it will be triggered as soon as you touch any knob on the screen. However, we're not interested in just any knob: we only want to respond to the first one. That's why we use an if-statement to check if the correct knob was used:

```
if LastKnob = 0
```

When the user twiddles a knob, the number of the knob is put into *LastKnob*. This way we can check for certain knobs whenever the *OnKnobChange* event is triggered. If multiple knobs are changed simultaneously, Mozaic will trigger separate *OnKnobChange* events for them, to make sure you can handle them all: no need to worry about events getting lost in the mayhem.

If the knob is indeed number 0, we do the following:

```
setting = GetKnobValue 0
SendMIDICC 0, 7, setting // send out CC#7 on channel 0
```

First we get the current value of knob 0, using *GetKnobValue*. Knobs conveniently use the range of 0-127, just like MIDI CC messages, so we can use its value immediately for sending out CC changes. That's what we do in the next line. *SendMIDICC* asks for a channel number (we take 0), a controller number (we take 7, which means volume) and a value. Instead of hard-coding a value there, we put *setting* into it. As a result: whatever the knob was set to, we will send out as a volume level. Easy-peasy!

Lesson 4: Using an LFO and a metronome to generate notes

LFOs are lots of fun. These slow waves can be used to breathe life into any sound or musical piece, and they are particularly useful in MIDI applications. In this lesson we're going to use an LFO to generate notes. We'll also learn how to set up the programmable 'metronome' to make things happen at tempo-synced intervals.

Let's say we want to play a note every 16th step whenever the AU host is playing; a very rudimentary generative music script (admittedly more interesting from a scripting perspective than from a musical perspective).

First let's look at the script…

```
@OnLoad
   SetMetroPPQN 4
   SetupLFO 0, 36, 96, no, 0.17
@End

@OnMetroPulse
   lfovalue = GetLFOValue 0
   SendMIDINoteOn 0, lfovalue, 100
   SendMIDINoteOff 0, lfovalue, 0, 500.0
@End
```

In the OnLoad event, we set up the metronome and the LFO we need. We want to send a note every 16th step, so we need to tell the metronome to notify us 4 times per beat (= 4 times per quarter note). This is what we do in the first line:

```
SetMetroPPQN 4
```

PPQN stands for "pulses per quarter note". So in a standard 4/4 time signature (which consists of 4 beats, or quarter notes) it will send pulse events 16 times per bar. Exactly what we need! If we'd set it to 3, we would get triplets. The nice thing about the metronome is that it will always automatically sync to the host's tempo (if you need something to happen at tempo-independent intervals, you can set up a timer instead).

> Tip: The maximum resolution of the metronome is 384. Be careful with high PPQN values for the metronome as it may result in higher CPU loads!

With the next line we start up an LFO. You can have a maximum of 16 LFOs running simultaneously, numbered 0-15, but here we'll only need one. Let's use number 0:

```
SetupLFO 0, 36, 96, no, 0.17
```

The *SetupLFO* command takes 5 parameters: the number of the LFO (0), the minimum value of the LFO (36), the maximum value of the LFO (96), whether we want to sync it to tempo (no), and the frequency: 0.17Hz. If tempo sync was enabled, the frequency would be set in cycles per bar.

The default LFO waveform is a sine, which we don't change here. But we could change the LFO wave using the *SetLFOType* command if we wanted to.

> Tip: You can choose from several different LFO waves. Using *SetLFOType* you can select: *Sine, Cosine, Square, Triangle, RampUp, RampDown, SH*. The last one is a sample & hold generator which can be interesting for randomized behavior.

We set up the metronome to notify us 4 times per quarter note. This is done using the *OnMetroPulse* event. Since the LFO is always running in the background, without requiring our attention, we can simply get its current value at any moment we like. The value will always be as accurate as possibly (typically that's sample-accuracy, depending on how the AU host is set up).

That's what we do inside the OnMetroPulse event handler:

```
lfovalue = GetLFOValue 0
SendMIDINoteOn 0, lfovalue, 100
SendMIDINoteOff 0, lfovalue, 0, 500.0
```

First we get the current value of LFO 0, using *GetLFOValue*. We put its value into *lfovalue*.
Next up, we send a MIDI Note On message (to be sent immediately) and we queue the accompanying Note Off message for half a second (500ms.) later.

SendMIDINoteOn and SendMIDINoteOff take 3 mandatory parameters, and an optional 4th one: channel number (we take 0), note number (in this case we put variable *lfovalue* there) and velocity values (100 and 0 respectively, in our example). All *SendMIDIxxxx* commands let you add an optional delay parameter; the time in milliseconds before the message is sent out.

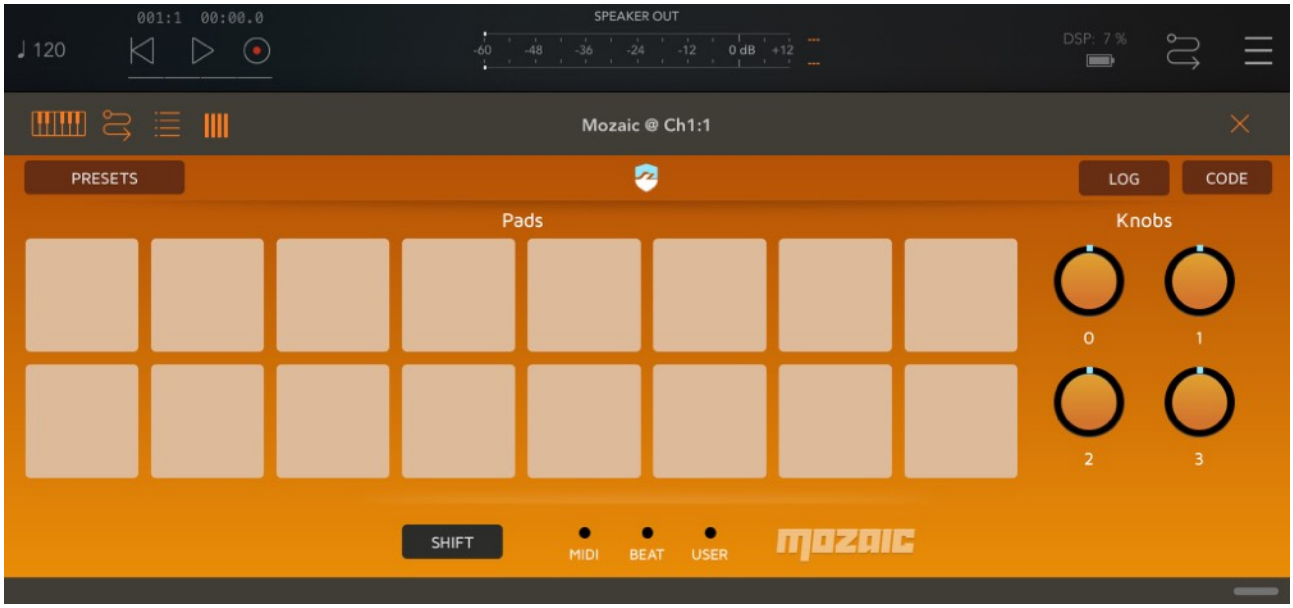
If you leave out the delay parameter, Mozaic will assume you want to send the MIDI event out immediately. In this case we use the delay parameter to queue a note off message 500ms. after the note on message. That's convenient, because it means we don't have to keep track of any notes that are playing!

> Tip: Don't forget to hit the PLAY button in your AU host or this script will not do much. Metronome events are only sent when the host's transport state is active!

That's it… your first generative music script! You can now add more LFOs, use *ScaleQuantize* to fit all notes into a proper scale and add a bunch of randomizations to turn this into something more fancy.

Lesson 5: Using an array to create a drumpad controller

Mozaic layout 2 has a nice layout with drumpads. Let's set these up so that they can control Ruismaker (or any other drum machine - as long as you can trigger it using MIDI notes).



Instead of using a bunch of if-else-statements to assign MIDI actions to pads, we're going to use an array variable this time, to demonstrate a different (and more effective) technique for mapping MIDI notes to different pads. Let's have a look at the script…

```
@OnLoad
   ShowLayout 2
   // these are the 8 notenumbers for Ruismaker drums
   notenum = [49, 51, 54, 56, 58, 61, 63, 66]
@End

@OnPadDown
   nn = notenum[LastPad]
   vel = LastPadVelocity
   SendMIDINoteOn 0, nn, vel
   SendMIDINoteOff 0, nn, 0, 500.0
@End
```

The first line in the *@OnLoad* event shouldn't be a big mystery to us anymore. It simply selects layout 2, which happens to be the layout with the 16 pads (and 4 knobs).

The next 8 lines are more interesting. Here we're going to use an array for the first time. Any variable in Mozaic can be used as an array. An array lets us address 1024 individual, numbered cells inside a single varable:

```
notenum[0][1][2][3][4][5]…[1023]
```

Inside this array we can store whatever we like: every cell is exactly like any other variable. The powerful thing is that we can reference each cell by its index number. We can use this to 'parametrize' our code:

```
v = Random 0, 1023
Log recordednotes[v]
```

If you see something like this for the first time, it may look like mindbending inception stuff. But it's incredibly powerful once you understand what's going on.

Essentially, in the snippet above we put a random value between 0 and 1023 into *v*, and then we use this value to pick a random cell from the array in *recordednotes* and print its value on the screen. If the array in *recordednotes* contained recorded MIDI notes, this would let you pick random notes from the recording.

You could rewrite this using if-else-statements, but you would need exactly a thousand of them to do the same thing we do in 2 lines here :-)

Back to our Ruismaker script. Here we use an array of 8 cells to store the MIDI note mapping of Ruismaker. If you're using another drum machine, you'll have to look up its note mapping and replace the Ruismaker values with your own.

The *OnPadDown* event is triggered whenever the user hits one of the pads on the screen. When the event handler is run for you, the number of the pad that was hit is put into *LastPad* and the velocity (based on how far from the center the pad was hit) is put into *LastPadVelocity*. We query these in the first two lines:

```
nn = notenum[LastPad]
vel = LastPadVelocity
SendMIDINoteOn 0, nn, vel
SendMIDINoteOff 0, nn, 0, 500.0
```

We do something interesting in the first line: we use the value of LastPad to reference a cell index of the array we set up in *OnLoad*: *notenum*. LastPad is a built-in function which will give us the number of the last pad which was hit by the user; i.e. the pad that triggered the event we're currently handling!

So after running the first line, *nn* will contain one of the note numbers mapped to our drum machine. Cool beans.

Next we put the velocity registered by the pad in *vel*. And in the last two lines we send out the note numbers (and the velocity) pretty much like we did in the last lesson.

We only put 8 values into the array, but there are 16 pads. So there is the real possibility of the user hitting a pad we haven't mapped to anything. We should add a check there to make sure we don't send out some undefined shizzle when the user hits a pad > 8. We could do something like this:

```
@OnPadDown
   if LastPad < 8
      nn = notenum[LastPad]
      vel = LastPadVelocity
      SendMIDINoteOn 0, nn, vel
      SendMIDINoteOff 0, nn, 0, 500.0
   endif
@End
```

Mozaic pads driving an instance of Ruismaker…

3. Understanding MIDI input

Welcome to 1986

MIDI is ancient by technology standards. Introduced in the mid ‘80s of the last century, it has hardly changed since it was presented alongside some (by now) vintage analog synthesizers. The fact that we still use it today is probably testament to how brilliant the original concept was.

However, when it was invented every digital bit was still expensive (in terms of storage, transmission and performance) so for reasons of optimization some aspects of the standard may feel a tad inconvenient and clumsy.

MIDI basics

MIDI timing
MIDI is transmitted as a series of events and commands. There are no timestamps in MIDI, so whenever an event comes in, it needs to be handled immediately. If you want something to happen, you send out the event at the moment when it needs to happen.

> Note: Mozaic lets you schedule events in the future (using an optional “delay” parameter in all MIDI functions). That’s not how MIDI works, so behind the scenes Mozaic keeps a to-do-list of events that need to be sent out, and it will send them out for you at exactly the right moment.

MIDI Channels, and channel numbering
To allow several MIDI devices (and apps) on a single MIDI chain the “MIDI channel” concept was introduced. The idea was that each device on the chain claims one or more channels for its own communication and ignores other channels (containing stuff that is intended for other devices). In the physical world, “MIDI THRU” ports on synthesizers let you connect multiple devices in a long chain. In software it still works more or less the same, except virtual MIDI cables are cheap so the channel concept isn’t as important anymore as it used to be. Still; it can be used to filter out MIDI messages that are meant for different applications - and multiple Mozaic plugins can be chained after each other.

On most devices and in most end user manuals Channels are referred to as 1-16 (there are a total of 16 channels available). Confusingly that’s not how MIDI works internally: when dealing with MIDI messages we count 0-15 instead. So that’s also what we’ll do in this guide. So while the drum channel is traditionally on Channel 10, all events for the drum channel are actually sent to channel 9 in code.

> Note: although MIDI manuals always talk about MIDI channels 1-16, they are internally numbered 0-15 instead, and that’s the convention this guide will follow!

Channel messages and system messages
There are roughly two kinds of MIDI events: *channel messages*, which are meant for one specific MIDI channel, and *system messages*, which are meant to be received by all channels.

Examples of channel messages: Note On, Note Off, Pitch Bend, Continuous Controller (CC) messages.
Examples of system messages: MIDI clock, Active Sensing, System Reset.

Most of the messages you'll come across using Mozaic will be channel messages, i.e. containing a channel number as part of their command code.

Anatomy of MIDI events
MIDI events are transmitted in packets of 1, 2 or 3 bytes. The first byte contains the command and (when appropriate) the channel information. Commands are simply a number, and the channel number is added to this number.

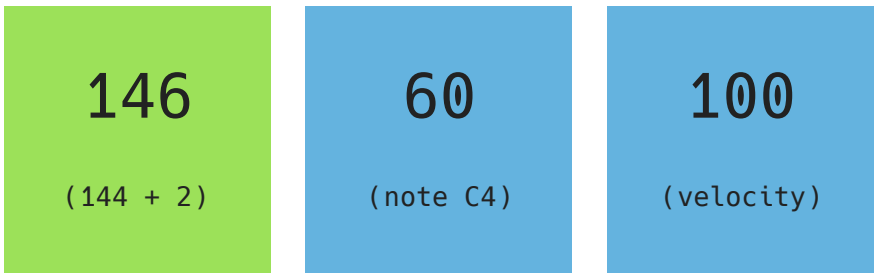
As you may know, a byte consists of 8 bits. The first 4 bits are used for the command, and the last 4 contain the channel number. Depending on the command, 1 or 2 extra bytes containing the interesting data for the event can follow the first byte.

Note messages and controller messages are 3 bytes long. A Clock message on the other hand doesn't contain any channel information and doesn't have any additional data. It is therefore only 1 byte long.

Some AU hosts do not allow transmission or reception of Sysex messages. If your script relies on Sysex, make sure your host supports it before testing.

Example: a MIDI Note On message
- A note on command has a base number of 144 (in hexadecimals this is 0x90)
- The channel number will be added to this
- This combination (command + channel) makes up the first byte
- The second byte contains the note number (0-127)
- The third byte contains the velocity of the note.

So if we want to send a Note On event on channel 2, for note C4, with a velocity of 100, we get the following 3 bytes:



If that looks difficult: sorry, there's not much we can do to simplify that. That's simply the nature of MIDI :-)

Hexadecimal numbers are typically used to describe MIDI commands. Although the concept of hexadecimals seems a bit esoteric if you've never used it, 'hex numbers' are very convenient for MIDI since MIDI's event numbering was fully designed around it. But let's not worry about that now.

Receiving MIDI in Mozaic

When a Mozaic plugin receives MIDI input, a number of things happen. If multiple MIDI events are received at the same time, they will be handled in sequence, in the order they are received in.

Important: as a script writer you only have to worry about one MIDI event at a time; because you will be notified about each event individually.

The incoming MIDI command is analyzed by the plugin, and depending on the command one or more events may be triggered for you in sequence from low level to specific:

@OnMIDIInput
This is the first event which will be called, and it will *always* be called for *any* MIDI command type. By checking the command byte in this single call all MIDI messages could be in theory be handled. However, for sake of efficiency and convenience, more specific event calls are also available…

@OnMIDINote
This is called for you when either a Note On or Note Off event is received. This is useful for handling note events where you want to perform the same actions on both Note On and Note Off events; such as matching to scales, filtering or forwarding ranges of notes, transposing notes, etc.

@OnMIDICC
Triggers when a MIDI CC event is received.

@OnMIDINoteOn
This event is called when a Note On command is received. By then you may already have received *@OnMIDIInput* and *@OnMIDINote.* Simply choose the handlers which are most convenient for you.

@OnMIDINoteOff
The Note Off counterpart for *@OnMIDINoteOn*. Should always be received after getting an *@OnMIDINoteOn*, or else there may be a risk of getting stuck notes.

Note that incoming MIDI notes with a velocity of 0 are automatically converted to NoteOff messages for you. You don't need to handle these in your script as your script will never get to see them.

@OnPedalDown
This is one specific instance of a MIDI CC message, so this one may sometimes follow after a call to *@OnMIDICC*. It is specifically offered to let you handle the event of a sustain pedal being triggered.

@OnPedalUp
This event is called when the sustain pedal is being released again.

Useful variables for your MIDI event handlers

Within your MIDI event handlers you'll probably want to know a bunch of things about the MIDI messages you're handling. For that reason Mozaic provides you with a number of read-only system variables which contain information you might find useful:

MIDIByte1, MIDIByte2, MIDIByte3
These are three byte-sized variables which contain the raw MIDI data which was received. Depending on the command they are filled with different things.

For example, if you're handling a Note message, the *MIDIByte2* contains the note number. However, in a MIDI CC message *MIDIByte2* contains the CC number. And in the case of a MIDI Clock message *MIDIByte2* doesn't contain anything useful at all.

```
@OnMIDIInput
   SendMIDIOut MIDIByte1, MIDIByte2, MIDIByte3 // this is a MIDI Thru
@End
```

When sending out MIDI data, Mozaic will double-check how many bytes should be transmitted and automatically resize the command accordingly. No need for us to worry about that ourselves.

To know what we're dealing with it would help to have easier access to the command information…

MIDICommand, MIDIChannel
This is essentially *MIDIByte1* (the command message) split into 2 useful parts:
- *MIDICommand* is the command message, but with the channel information removed (set to 0).
- *MIDIChannel* is channel number

This is particularly useful because we can now easily check the command, without having to worry about channel data getting in our way:

```
if MIDICommand = 144
  // this is a Note On command (in hex: 0x90)
else if MIDICommand = 128
  // this is a Note Off command (in hex: 0x80)
endif
```

Simply add MIDIChannel to MIDICommand to recreate the original command byte:

```
if MIDIByte1 = ( MIDICommand + MIDIChannel )
  Log {Yeah, this should be identical!}
else
  Log {You will never see this text}
endif
```

MIDINote, MIDIVelocity
If you know you're handling a MIDI Note message (On or Off) you can use these two read-only variables to easily do interesting things with the note data.

```
if MIDINote < 64 and MIDIVelocity > 100
  // we're only interested in strong notes in the lower half of the key range
  SendMIDIThru
endif
```

To show some of these things in use, here's a working script for a basic MIDI monitor. Note that the commands in this script are listed in hexadecimals. As you can see, the command numbers look less random in hex than in normal (decimal) numbers, don't they?

```
@OnMidiInput

  if MIDICommand = 0x90
    Log {Note On: }, MIDINote, {velocity: }, MIDIVelocity

  elseif MIDICommand = 0x80
    Log {Note Off: }, MIDINote

  elseif MIDICommand = 0xB0
    Log {CC change: }, MIDIByte2, {value: }, MIDIByte3

  elseif MIDICommand = 0xD0
    Log {Aftertouch amount: }, MIDIByte2

  else
    Log {Command: }, MIDICommand
  endif

  SendMIDIThru
@End
```

Although you may shudder at the thought of using hexadecimal numbers, it is really recommended to at least use them when referring to MIDI commands (0x90, 0x80, 0xB0, etc.), because this will make reading your scripts a lot easier - especially for others.

4. Interacting with the GUI

Mozaic lets users interact with our scripts using a convenient plugin user interface. All controls on the UI, such as knobs, pads and XY controllers are already hooked up and can be instantly accessed by your scripts. They also generate convenient events when the user changes their value, so you can handle any interaction immediately.

Layout options

There are 5 different layouts. Some controls appear on multiple layouts. E.g. several layouts have knobs on them; and Knob 0 on the first layout is actually the same as Knob 0 on other layouts. So when handling events you it doesn't necessarily matter to know which layout is currently active. A call from Knob 0 always comes from the first knob on the screen. Note that sliders (on layout 3) are treated as knobs. You can access sliders and their labels using the same functions as you would access a knob.

There are:
- 22 knobs/sliders (numbered 0-21)
- 16 trigger pads (0-15)
- 1 XY pad
- 1 Shift Button

These are the 5 layout options:

Layout 0: Mix

Layout 1: Knobs

Layout 2: Pads

Layout 3: Sliders

Layout 4: Minimal

You can choose a layout in your script using the *ShowLayout* command:

```
ShowLayout 4 // display the minimal UI with the description panel
```

> The states of all knobs and controls (even those that are not on the screen right now) are saved and restored as part of AU state saving.

Knobs (and sliders)
There are 22 knobs/sliders available in total. But only Layout 1 shows all of them on the screen simultaneously. The value range is 0-127, but unlike MIDI CC values high-resolution fractional values are possible. By using this range you can immediately apply the setting of knob for sending out CC values without any conversion of scaling required.

```
@OnLoad
  Labelknob 0, {Try Me!}
  Labelknob 1, {Watch me...}
@End

@OnKnobChange
  if LastKnob = 0
   v = GetKnobValue 0
   SetKnobValue 1, v
  endif
@End
```

XY Pad
The XYPad is like having 2 sliders packed into one. Both axes of the pad have a 0-127 value range. When either of the two values changes, a single *@OnXYChange* is generated. Always assume that both values have changed together; in practice it's also really tricky, if not impossible, to only change a single axis on the pad. That's not what XY pads were designed for :-)

```
@OnXYChange
  // let's compute the distance from the center of the pad
  delta1 = Abs ( GetXValue - 64 )
  delta2 = Abs ( GetYValue - 64 )
  pyt = Sqrt ( ( delta1 * delta1 ) + ( delta2 * delta2 ) )
  Log {Pythagoras says: }, pyt
@End
```

Trigger Pads

The 16 pads can obviously be used as triggers, but also be 'misused' to display basic status information. Each of them can be made to light up via script: there are functions for making them flash, but also for switching the background LED on and off (i.e. a latch mode) and changing their color tint. You can also put labels on them.

When the user triggers a pad by tapping on it, a velocity is derived from it (by looking at how close to the center of the pad the tap was placed). The following example shows how to use these mechanisms. We switch Latch mode on the pad on and off when the user taps it with a high velocity (so: near the center of the pad).

```
@OnLoad
  ShowLayout 2 // all pads
  // initialize all latch states to off
  for pad = 0 to 15
    LatchPad pad, NO
    LabelPad pad, pad + 1
  endfor
@End

@OnPadDown
  if LastPadVelocity > 100
    // invert the state of the pad that was tapped
    LatchPad LastPad, NOT (PadState LastPad)
  endif
@End
```

If you want to try how the Flash function for pads works, add the following script and hit the Shift button a few times:

```
@OnShiftDown
  FlashPad (Random 0, 15)
@End

@OnShiftUp
  FlashUserLed
@End
```

5. Using LFOs in Mozaic

One of the convenient features of Mozaic is the fact it has up to 16 easily accessible LFOs built right into the plugin. The LFOs are free-running and can be optionally synced to the AU host tempo.

Using LFOs is absolutely trivial:

1. Choose a waveform
2. Set up the LFO (tell Mozaic the frequency of the wave, whether you want to sync it, the range of values you need, etc.)
3. Get values from the LFO whenever you need them
4. Rejoice

Each of the 16 available LFOs can be configured separately. The LFOs automatically run in the background and always return a very precise value (phase calculations are done at sample accuracy).

There are several waveforms to choose from:

- Sine,
- Cosine,
- Square,
- Triangle,
- RampUp,
- RampDown,
- SH

To illustrate the use of LFOs, here's an example script which lets you change the frequency of the LFO using a knob on the screen. The value of the LFO is used to make the XY pad move up and down to visualize the LFO…

```
// set up
@Onload
  LabelKnob 0, {LFO Speed}
  SetupLFO 0, 0, 127, no, 1.0
  SetTimerInterval 100
  StartTimer
@End

//here we take the current value of the LFO and make the XY pad dance
@OnTimer
  v = GetLFOValue 0
  SetXYValues 64, v
@End

// adjust the LFO speed when we tweak the knob
@OnKnobChange
  if LastKnob = 0
    k = GetKnobValue 0
    k = k / 64 + 0.1
    SetupLFO 0, 0, 127, no, k
  endif
@End
```

6. Remembering note states

Many of your Mozaic scripts will revolve around managing and handling incoming MIDI notes. Contrary to most other MIDI concepts a MIDI Note consists of two separate parts: a Note On event and a Note Off event. More often than not, you'll want to keep track of some information between these two events.

For example: imagine a script that takes incoming notes and sends them out to a random channel.

If you don't remember what channel you chose to send a note to, you'll have a problem by the time you receive the Note Off event. If you don't want to get stuck notes you need to remember the 'randomized channel' you used for Note On and use that for your Note Off. The obvious answer is to use an array for that, but you either need a 2-dimensional array (for dealing with up to 16 channels x 128 notes) or a really large array which is much bigger than Mozaic allows. Here's what we want to do:

On NoteOn > Choose a random channel > Store random channel > Send note out to random channel
On NoteOff > Retrieve random channel for this note > Send out Note Off to random channel

A Virtual Locker Room
To enable this Mozaic has introduced the NoteState matrix. This matrix is like a big locker room, with a locker for each possible MIDI note: 16 channels x 128 notes. At any moment you can store a value in a locker and get it out whenever you need. Each locker is labeled by its channel and note number and can be accessed with the *GetNoteState* and *SetNoteState* functions. Here's our example in actual code:

```
@OnLoad
  ResetNoteStates // initialize the NoteState array; good practice
@End

@OnMidiNoteOn
  r = Random 0, 15 // make a random MIDI channel
  SendMIDINoteOn r, MIDINote, MIDIVelocity // send note to random channel
  SetNoteState MIDIChannel, MIDINote, r // remember the channel
@End

@OnMidiNoteOff
  r = GetNoteState MIDIChannel, MIDINote // retrieve the randomized channel
  SendMIDINoteOff r, MIDINote, MIDIVelocity // use the randmized channel
@End
```

So we use the actual *MIDIChannel* and *MIDINote* as a locker ID to store the randomized channel, because this channel and note number will also be received in the corresponding Note Off event.

This technique can be used for many different note-related tasks. The added benefit of using this mechanism is that it offers a consistent approach across different scripts, making scripts that use it more readable and predictable. When you come across these functions, you know what they'll do.

The contents of the Note State matrix are not persistent across sessions; meaning states are not saved in project files and presets. This wouldn't make sense anyway, because actual MIDI states will have changed by the time you reload your script.

7. Sysex

System Exclusive messages (aka "Sysex") occupy an obscure corner of the MIDI protocol. They were devised to allow manufacturers to send any kind of non-standard data blobs over a MIDI cable. In the pre-USB era, if you wanted to dump a sample into a sampler, or get some user-presets out of a hardware synth, you would often use Sysex to do it. Even today many hardware instruments and effect boxes use Sysex to communicate with the outside world.

Sysex handling is not without challenges:
- Sysex messages can have any size or length
- Some devices require complicated checksum calculations
- Sending too many (long) messages can quickly choke a MIDI chain
- Every manufacturer uses a different format for the enclosed data (and documentation can be lacking, difficult to find or highly technical)

As of version 1.2 Mozaic has support for Sysex input and output. There are some limitations and things to keep in mind:
- Mozaic can only send/receive messages of up to 1024 bytes in length
- You can not queue outgoing Sysex messages; they will be sent immediately (as soon as all events of the current time slice are finished)
- Sysex handling is completely separate from the 'regular' MIDI event handling. I.e. incoming Sysex will not trigger an OnMidiInput event.
- Unless a script explicitly handles them, Sysex messages are not received or passed through Mozaic scripts
- The total size of messages to be sent out in one go cannot exceed 16Kb

---

Sysex messages

A Sysex message can have any format. The only commonality is that all messages start with a byte *0xF0* and end with a byte *0xF7.* Everything in between these two bytes is the Sysex data which you can freely access and manipulate with Mozaic. This sysex data typically contains several things such as manufacturer and device IDs, checksums, etc. As a consequence, it's hard to make sense out of a Sysex message without the original documentation of the device you're communicating with.

For example, to set the filter cutoff on a vintage Juno 106 to 64, you'd send the following Sysex message:

```
0xF0  0x41  0x32  0x00  0x05  0x40  0xF7
```

This is a relatively simple message, and its data (so not counting the start and end-bytes) is only 5 bytes long. The first 0x41 byte signifies this is a Roland message (there is no device ID specified) and the remaining 4 bytes specify the message type, channel, parameter and value.

In Mozaic, you never need to worry about the start and end bytes (0xF0 and 0xF7). Only the Sysex data itself is of importance to us. We use arrays to receive, contain and manipulate the data.

Sending Sysex messages

To send the Juno 106 message with Mozaic we first have to put the data into an array variable and then call the SendSysex command. In the example we send the value of Knob 0 as a cutoff value to the Juno:

```
@OnKnobChange
  value = GetKnobValue 0
  data = [0x41, 0x32, 0x00, 0x05, value]
  SendSysex data, 5
@End
```

That's easy enough. First we get the value of the Cutoff parameter we want to send from Knob 0. Then we fill an array variable (named '*data*') with our Sysex format and put the desired value in the fifth byte of the array. Finally we send out the message, specifying the array which contains our data and the length of the message content (5 bytes).

Sending Sysex messages with a checksum
Not wanting things to be too simple for us, some manufacturers require checksums to let devices check the validity of Sysex messages (i.e. to prevent corrupted data from causing problems). Typically, such a checksum is the last byte - at the end of the Sysex data.

You could calculate these checksums as part of the script, but it's not trivial. To help with the most common cases, Mozaic can calculate the checksums for you when sending the message.

The following checksum algorithms are currently supported:
0 - No checksum (default)
1 - Roland/Boss
2 - Fractal Audio (AxeFX)

To help us, the *SendSysex* command has 2 optional parameters which let us specify which checksum algorithm to use (if any) and the index in the data array at which to start the calculation:

```
SendSysex <array>, <length>, <checksum algo>, <checksum start index>
```

If checksum algorithm and start index are not specified, Mozaic will assume you don’t want to send a checksum; so it’s safe to leave them out for the sake of code-readability in that case.

An example with using checksum
Let's look at an example. We're going to send a 'checksummed' Sysex message to a Boss Katana. The format of its data consists of several parts:

1 byte	manufacturer ID
5 bytes	device and model ID
1 byte	message type (i.e. is this message a query or a settings update?)
4 bytes	parameter address
1-n byte(s)	updated parameter value (this can be more than 1 byte)
1 byte	checksum

So most Katana settings messages are 13 bytes, excluding the start and end bytes, but including the checksum byte. If you're changing a parameter setting the first 7 bytes will always be the same. For example, if we want to change the reverb type we could use the following script:

```
@OnKnobChange
  rev = GetKnobValue 0 // get reverb type
  data = [0x41,0x00,0x00,0x00,0x00,0x33,0x12,0x60,0x00,0x12,0x14, rev ,0x00]
  SendSysex data, 13, 1, 7
@End
```

The first 7 numbers in the data array are the fixed ID and message type bytes. Next, there's the 4 byte address for "Reverb Type". Then follows our new value for the Reberb Type parameter and finally we reserve one byte for the checksum (the 0x00 is just a placeholder for us and doesn't mean anything).

The checksum is calculated when we send the message and will automatically replace the last byte. For this device model we use the Roland/Boss checksum algorithm (1). According to the documentation of the device the checksum calculation starts with the parameter address, which is the 8th byte. Counting starts a *data[0]*, so we can find the 8th byte at *data[7]*.

Hence we end up with the line:

```
SendSysex data, 13, 1, 7
```

Unfortunately for us, each device (even coming from the same manufacturer) will have different Sysex format. So if you want to support Sysex message there's no way to avoid doing lots of manual diving…

Receiving Sysex

The standard *OnMIDIInput* event does not respond to Sysex messages. Instead, there's a separate event to capture incoming Sysex data: *OnSysex*. When a Sysex message is received, we can load its data into an array for processing. Note that messages > 1024 bytes are ignored by Mozaic.

Have a look at the follow example script:

```
@OnSysex
  ReceiveSysex data
  length = SysexSize
  Log {The received Sysex data is }, length, { bytes long.}
  SendSysexThru
@End
```

Let's go through this script to see what's going on...

- The OnSysex event is invoked when a Sysex message of less than 1024 bytes is received
- First we tell Mozaic to put the incoming Sysex data into a variable The *ReceiveSysex* command does this for us. In this case we let it create a variable '*data*' and tell it to load it up with the Sysex contents.

- Next we query how big the incoming Sysex message is. Note that the start/end-bytes are excluded (and also aren't received with ReceiveSysex)
- Finally (after logging some stuff) we tell Mozaic to send through the original incoming message unchanged.

Checking sysex headers

Often you'll want to check the header of a Sysex message to verify you've captured a message en route to/from a specific device. Here's an example script to accomplish this.

```
@OnSysex
  ReceiveSysex data
  header = [ 0x13, 0x37, 0xB0, 0x55 ] // example header to check for
  found = YES

  // check the header...
  for i = 0 to 3
    if data[i] <> header[i]
      found = NO
    endif
  endfor

  if found
    // Do something with this Sysex
  else
    // These are not the bytes you're looking for
  endif
@End
```

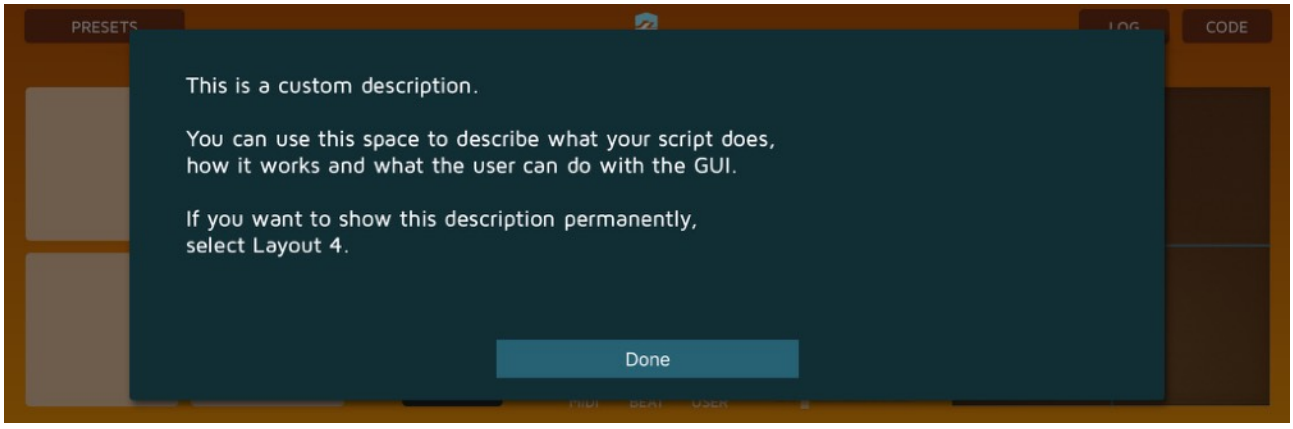
8. Overview of Mozaic Events

Here you'll find an overview and description of all event triggers Mozaic offers. For each event there can be one handler in your scripts (so you can't have more than one handler referring to the same event).

When multiple related events are triggered, they are ordered from generic to specific. For example: when a MIDI note is received, it triggers three events in sequence: first *OnMIDIInput*, then *OnMIDINote,* and finally *OnMIDINoteOn*. You don't need to handle all of them, just pick the one(s) you need for your use case.

`@OnAUParameter`	Triggered when one of the AU User Parameters is changed by the Audio Unit host. There are 8 AU Parameters which can be directly accessed by AU hosts (labeled "AU Parameter 0" through "AU Parameter 7"). These are useful for cases which require accuracy because they have a much higher resolution than regular MIDI CC values.
`@OnHostStart`	Triggered when the AU host starts playing.
`@OnHostStop`	This event handler is called when the AU host's transport is stopped.
`@OnKnobChange`	Triggers when the user changes a value on one of the knobs. There are 22 knobs which can be accessed, although not every layout shows all knobs. Check *LastKnob* to find out which knob was changed.
`@OnLoad`	Called when a script is uploaded, loaded as a preset, or when an AU host restores the plugin's state (e.g. when loading a project containing a Mozaic script). This is de perfect event for performing all the necessary initialization actions in your script, such as setting variables to their initial values and selecting the right layout for your GUI.
`@OnMetroPulse`	Triggers when the metronome sends a pulse. You can set the number of metronome pulses per beat you want to trigger using *SetMetroPPQN*. Pulses are counted using *CurrentMetroPulse*. Counting is automatically restarted when a new bar starts.
`@OnMidiCC`	This event is called when the plugin receives a MIDI CC event. *MIDIByte2* contains the CC controller number, and *MIDIByte3* contains the value that was received.
`@OnMidiInput`	Called when any sort of MIDI Input is received. MIDIByte1, MIDIByte2, MIDIByte3, MIDICommand and MIDIChannel can be used to determine the type of event, and any interesting extra information.
`@OnMidiNote`	Triggered when MIDI Note events are received. This can be either Note On or Note Off events. This is a useful event for performing actions on notes where you want Note On and Note Off messages to be treated the same way; e.g. for quantizing to a certain musical scale, transposition or filtering out ranges of notes.
`@OnMidiNoteOff`	This event is called when a Note Off event is received. You can check *MIDIChannel*, *MIDINote* and *MIDIVelocity* for the relevant details.
`@OnMidiNoteOn`	Triggers when a Note On event is received. You can check *MIDIChannel*, *MIDINote* and *MIDIVelocity* for the relevant details.

`@OnNewBar`	This event is called when the Audio Unit host is playing and the playback head reaches a new bar. Timing is sample-accurate. You can access the current bar and beat values using *HostBar* and *HostBeat*. This event will not be triggered when the host is stopped, even when the playback head is on the start of a new bar.
`@OnNewBeat`	This is like *OnNewBar* above, except it is called every time the playhead reaches a new beat. The number of beats per bar (which is time-signature dependent) can be queried using *HostBeatsPerMeasure*.
`@OnPadDown`	Triggered when the user touches a pad on the screen. Check *LastPad* to see which pad was tapped. Use *LastPadVelocity* to derive the velocity (=how close to the center) of the tap.
`@OnPadUp`	Called when the user lets go of a pad. Use *LastPad* to see which pad was released.
`@OnPedalDown`	This event is triggered when the user pushes down a sustain pedal. This event is agnostic of MIDI channels (multiple pedals could in theory be hooked up to different MIDI channel) so if you need to make a distinction between different pedals, use the *MIDIChannel* value to do so.
`@OnPedalUp`	Triggered when the user lets go of a sustain pedal.
`@OnShiftDown`	Is called when the user touches the on-screen Shift button.
`@OnShiftUp`	Called when the user lets go of the on-screen Shift button.
`@OnSysex`	Invoked when a Sysex message (<1024 bytes) is received.
`@OnTimer`	Called at regular timer intervals, defined by the timer. The timer runs independently of AU host tempo, so its interval (sample-accurate in milliseconds) is always exactly the same. Use *SetTimerInterval* to set the desired interval time. The timer ignores the host play state. Instead you can use *StartTimer*, *StopTimer* and *ResetTimer* to control its active state.
`@OnXYChange`	Notifies the script that the user changed (or is changing) the XY pad values. Use *GetXValue* and *GetYValue* to get query the current values of the XY pad.
`@Description`	This is not an event like the other events. Instead, you can type a descriptive text in the lines between *@Description* and *@End* which will be shown to your users when they load the script. It's also displayed in a text box on Layout 4.



9. Mozaic Commands and Functions

This is a complete alphabetical documentation of all commands and functions available in Mozaic. There's also a function overview ("cheat sheet"), sorted by category, in the next chapter.

Abs

`<result> = Abs <value>`

Description: returns the absolute value of <value>

Parameters and output:

`<result>` the absolute value of <value>

`<value>` any input value or variable

Example:

```
delta = Abs (x - y)
Log {The difference between x and y is: }, delta
```

Clip

`<result> = Clip <value>, <min>, <max>`

Description: limits the value <value> to the range <min> to <max>. If the input value is in this range it is returned unchanged. If the input value is smaller than the minimum value then the minimum value is returned instead. If the input value is greater than the maximum value the maximum value is returned instead.

Parameters and output:

`<result>` a value limited to the range specified by <min> and <max>

`<value>` any input value or variable

`<min>` minimum value for the clipping range

`<max>` maximum value for the clipping range

Example:

```
velocity = MIDIByte3 + 32
velocity = Clip velocity, 0, 127
SendMIDICC MIDIChannel, 7, velocity
```

ColorPad

`ColorPad <padnum>, <colornum>`

Description: Adds a color tint to the pad with number <padnum> as if it were a backlit rubber drumpad. There are 8 tints available:
0 - Soft rubbery white (default)
1 - Red
2 - Yellow
3 - Green
4 - Light blue
5 - Lavender-purple
6 - Violet
7 - Magenta

Parameters and output:

`<padnum>` the number of the pad you wish to color (0-15)

`<colornum>` the number of the tint you want to apply (0-7)

Example:

```
for i = 0 to 15
  ColorPad i, (random 0, 7)
endfor
```

ConfigureMPE

`ConfigureMPE <lowerzone>, <upperzone>`

Description: sends a so-called MPE Configuration Message (MCM) to identify your plugin as an MPE controller. Both available zones can be specified in this command, although MPE compatible devices and synths typically only use the “lower zone”. When an MCM message is sent with a value of 0 for a zone, that zone is disabled as an MPE zone.

Parameters and output:

`<lowerzone>` the number of MIDI channels you want to allocate to the lower zone. If set to 0, the lower zone will be disabled. If set to a non-zero value, MIDI channel 0 will be the lower zone’s master channel and the MPE channels will count from 1 upwards.

`<upperzone>` the number of MIDI channels you want to allocate to the upper zone. If set to 0, the upper zone will be disabled. If set to a non-zero value, MIDI channel 15 will be the upper zone’s master channel and the MPE channels will count from 14 downwards.

Example:

```
ConfigureMPE 15, 0 // reserve all channels for the MPE lower zone
ConfigureMPE 0, 0 // disable all MPE zones
```

CopyArray

`CopyArray <sourcevar>, <destvar> [, <numcells>]`

Description: Copies (part) of an array into another array. Optionally the number of cells to be copied can be specified. Since default arrays can be quite big, it is recommended for performance reasons to copy as few cells as possible.

Parameters and output:

`<sourcevar>` The source array variable

`<destvar>` The destination array variable

`<numcells>` The number of cells to be copied. This is an optional parameter, but it's recommended to specify this value to optimize performance.

Example:

```
a = [0,1,2,3,4]
CopyArray a, b, 5 // copy the first 5 cells of a into b
CopyArray a, b[5], 5 // copy the same cells also into the next 5 cells of b
```

Cos

`<result> = Cos <value>`

Description: returns the cosine of the input value.

Parameters and output:

`<result>` the cosine of the input value

`<value>` input value

Example:

```
v = Cos ( phase * ( 3.14159 * 2 ) )
```

CurrentMetroPulse

`<result> = CurrentMetroPulse`

Description: returns the current metronome pulse. The pulse counter is reset to 0 when the host enters a new bar.

Parameters and output:

`<result>` the current metronome pulse counter

Example:

```
if ( CurrentMetroPulse % 2 ) = 0
  Log {only trigger on even pulses!}
endif
```

CustomScale

`CustomScale <c>, <c#> <d>, <d#>, <e>, <f>, <f#>, <g>, <g#>, <a>, <a#>, <b>`

Description: sets the active scale to a custom user-defined scale. When specifying this scale, assume the rootnote is C. For each note in the octave you can speficy whether it's part of the scale or not, using boolean values. Once specified you can use *SetRootNote* to change the root note of the scale.

Parameters and output:

`<c>-<b>` 12 boolean values indicating whether each of these notes is part of the scale

Example:

```
// minor pentatonic
CustomScale yes, no, no, yes, no, yes, no, yes, no, no, yes, no
SetRootNote 2 // D
newnote = ScaleQuantize MIDINote // quantize the incoming midi note
```

Dec

`Dec <var> [,<min>]`

Description: decreases the variable <var> by 1. You can optionally specify a minimum value. If unspecified, the minimum value is 0.

Parameters and output:

`<var>` the variable to be decreased. If the variable is greater than the minimum value, 1 will be subtracted. Otherwise it will be set to the minimum value.

`<min>` optional minimum value. If this parameter is left out 0 will be used instead.

Example:

```
repeat
  Log counter
  Dec counter
until counter = 0
```

Div

`<result> = Div <value1>, <value2>`

Description: performs an integer-division on value1 and value2.

Parameters and output:

`<result>` the result of value1 divided by value2, with the fractional part discarded

`<value1>` division denominator

`<value2>` division numerator

Example:

```
i = Div 10, 3 // result of 10 div 3 = 3
```

Exit

`Exit`

Description: immediately exits the current event handler without executing any further parts of its code. This command can be called in the middle of a loop, etc.

Example:

```
for i = 0 to 500
  if (Random 0, 10) = 5
    Exit
  endif
endfor
```

Exp

`<result> = Exp <value>`

Description: the Exp function returns the result of *e* raised to the power of <value>

Parameters and output:

`<result>` *e* to the power of <value>

`<value>` power value

Example:

```
x = Exp z // get the exponential value of z
```

FillArray

`FillArray <var>, <value> [,<numcells>]`

Description: Fills all cells of the array specified by <var> with <value>. Note that this Mozaic function will create the specified variable if it hasn't been created/initialized yet.

Parameters and output:

`<var>` variable to be filled

`<value>` value which will be put into the cells of the variable

`<numcells>` optional: the number of cells to be filled. Smaller numbers will result in better performance.

Example:

```
FillArray recordbuffer, 0 // reset all cells in the array to 0
FillArray a[64], -1, 64 // fill 64 cells with -1, starting at index [64]
```

FlashPad

`FlashPad <pad>`

Description: Flashes the pad with the number <pad> on the screen. If a layout is chosen which has no pads, or if the pad is currently latched nothing visible will happen.

Parameters and output:

`<pad>` Number of the pad that will flash. There are 16 pads, numbered 0-15, in total.

Example:

```
for pad = 0 to 15
  FlashPad pad
endfor
```

FlashUserLed

`FlashUserLed`

Description: Flashes the LED on the GUI labeled “USER”.

Example:

```
@OnTimer
  FlashUserLed
@End
```

GetAUParameter

`<result> = GetAUParameter <parameter>`

Description: returns the current value of the AU Parameter with number <parameter>. There are 8 user parameters available, numbered 0-7. Their range is 0-127, which makes them nicely compatible with MIDI CC. However, the resolution of these parameters is a 32bit floating point value, instead of the traditional 7 bits of MIDI CC.

Parameters and output:

`<result>` The current value of AU parameter number <parameter>

`<parameter>` The number of the parameter you want to query (0-7)

Example:

```
p = GetAUParameter 0
Log p
```

GetKnobValue

```
<result> = GetKnobValue <knob>
```

Description: returns the current value of the on-screen knob with number <knob>. There are 22 knobs in the plugin, although they may not all be present on the currently showing GUI layout. The range of the knobs is 0-127, to make them instantly compatible with MIDI CC, but their values have full floating point accuracy.

Parameters and output:

`<result>` The current value of knob number <knob>

`<knob>` The number of the knob you want to query (0-21)

Example:

```
value = GetKnobValue 0
value = TranslateCurve value, 0.33, 0, 127 // make response-curve non-linear
SendMIDICC 0, 15, value
```

GetLFOValue

```
<result> = GetLFOValue <lfo>
```

Description: returns the current value of the LFO with number <lfo>. There are 16 LFOs in the plugin. The range of the LFOs can be set using *SetupLFO.* Mozaic LFOs are free running and continuously run in the background. The value returned by GetLFOValue is always the most accurate value for that point in time.

Parameters and output:

`<result>` The current value of knob LFO <lfo>

`<knob>` The number of the LFO you want to query (0-15)

Example:

```
@OnMetroPulse
  value = GetLFOValue 0
  Log value
@End
```

GetNoteState

```
<result> = GetNoteState <chan>, <note>
```

Description: returns the value which was stored for note <note> in channel <chan>. The NoteState matrix is a 2-dimensional array, a virtual 'locker room', with a cell for each possible note which lets you store an arbitrary value and retrieve it when needed.

For example when you manipulate an incoming note in a NoteOn handler, you can then save some useful information in the note state matrix and retrieve that when the NoteOff event for that note is triggered.

The value can be numerical or boolean.

Parameters and output:

`<result>` The value which was stored for this note/channel combination

`<chan>` The channel number of the note (0-15)

`<note>` The note number (0-127)

Example:

```
@OnMidiNoteOn
  // store a random number for this note
  SetNoteState MIDIChannel, MIDINote, (Random 0, 5)
@End

@OnMidiNoteOff
  // recall the value we stored
  v = GetNoteState MIDIChannel, MIDINote
  Log {We stored: }, v
@End
```

GetXValue, GetYValue

```
<result> = GetXValue
<result> = GetYValue
```

Description: return the current values of X-axis and the Y-axis of the XY-pad respectively. Both can be queried independently. Depending on the current layout the XY-pad can be on the screen or not, but the X and Y values can always be retrieved regardless.

Parameters and output:

`<result>` The current value of either the the X or the Y axes of the XY pad. Values are in the range 0-127

Example:

```
@OnXYChange
  delta = Abs (GetXValue - GetYValue)
  SendMIDICC 0, 15, delta
@End
```

GetXYMorphValue

```
<result> = GetXYMorphValue <topleft>, <topright>, <bottomleft>, <bottomright>
```

Description: Applies a bilinear interpolation to the current position on the XY pad, given 4 values assigned to the four corners of the pad. In other words: you can assign a value to each corner on the XY Pad and 'morph' between those by changing the X and Y coordinates on the screen.

Parameters and output:

`<result>` The interpolated value based on the values assigned to the corners of the XY Pad

`<topleft>` Value assigned to top-left corner of the XY Pad. Can be any value.

`<topright>` Value assigned to top-right corner of the XY Pad. Can be any value.

`<bottomleft>` Value assigned to bottom-left corner of the XY Pad. Can be any value.

`<bottomright>` Value assigned to bottom-right corner of the XY Pad. Can be any value.

Example:

```
@OnXYChange
  tl = 30
  tr = 66
  bl = 80
  br = 127
  value = GetXYMorphValue tl, tr, bl, br
  Log value
@End
```

HostBar, HostBeat

```
<result> = HostBar
<result> = HostBeat
```

Description: return the current bar/measure and beat reported by the AU host. If the host is not in playback mode, the reported values will represent the static position of the playhead.

Parameters and output:

`<result>` the current bar and beat reported by the AU host.

Example:

```
@OnNewBeat
  Log HostBar, HostBeat
@End
```

HostBeatsPerMeasure

```
<result> = HostBeatsPerMeasure
```

Description: return the current beats per measure (redived from the time signature) reported by the AU host. Note: not all hosts report this figure correctly.

Parameters and output:

`<result>` the current beats per measure, as reported by the AU host.

Example:

```
var0 = HostBeatsPerMeasure
Log {Beats per measure, according to the host: }, var0
```

HostRunning

`<result> = HostRunning`

Description: returns whether the host is currently in playback mode. Use *@OnHostStart* and *@OnHostStop* instead if you want to catch the exact moment this state changes.

Parameters and output:

`<result>` the current playback state of the host's transport.

Example:

```
if HostRunning = YES
  Log {The host is in playback mode!}
endif
```

HostTempo

`<result> = HostTempo`

Description: returns the current tempo reported by the host. Can have a fractional part (e.g. 112.4)

Parameters and output:

`<result>` the current tempo reported by the host

Example:

```
@OnNewBeat
  if HostTempo <> lasttempo
    Log {The tempo has changed since the last beat.}
  endif
  lasttempo = HostTempo // store the tempo for next time
@End
```

Inc

`Inc <var> [,<max>]`

Description: Increases the variable <var> by 1. You can optionally specify a maximum value. If unspecified, the maximum value is 65535.

Parameters and output:

`<var>` the variable to be Increased. If the variable is lower than the maximum value, 1 will be subtracted. Otherwise it will be set to the maximum value.

`<max>` optional maximum value. If this parameter is left out 65535 will be used instead.

Example:

```
@OnNewBeat
  Inc beatcounter
  Log beatcounter, {beats passed...}
@End
```

InScale

`<result> = InScale <notenum>`

Description: checks whether the note number <notenum> is part of the currently active scale (also taking in account the current root note setting). Returns YES (or true) if so, and NO (or false) if not. Use *ScaleQuantize* to force the note in scale.

Parameters and output:

`<result>` A boolean value (YES/true or NO/false) depending on whether <notenum> is part of the active musical scale.

`<notenum>` the number of the note to check

Example:

```
@OnMidiNoteOn
  c = InScale MIDINote
  if c = NO
     Log {This note is not in the scale}
  endif
@End
```

LabelKnob

`LabelKnob <knob>, {label}, <value>, ..., <value>`

Description: changes the label of the on-screen knob <knob>. Any number of strings, variables and values can be listed to create a label-string. There are 22 knobs in Mozaic, numbered 0-21, but not all may be visible on every layout. This command is not to be confused with *LabelKnobs*, which changes the title above the knobs section on the screen. Labels which are too long will be truncated on the screen (…).

Parameters and output:

`<knob>` The number of the knob to relabel. This number has to be in the range of 0-21

`{label}` text-string containing (part of) the new label of the knob.

Example:

```
@OnLoad
  LabelKnob 0, {Volume (}, volume_level, {)}
  for i = 1 to 6
    LabelKnob i, {Knob }, i
  endfor
@End
```

LabelKnobs

`LabelKnobs {label}, <value>, ..., <value>`

Description: Changes the title label above the knobs section on the screen. This command is not to be confused with *LabelKnob*, which changes the labels of individual knobs.

Parameters and output:

`{label}` text-string containing the new label of the title. If title is too long it will be truncated (…).

Example:

```
@OnLoad
  LabelKnobs {Filter Controls}
@End
```

LabelPad

`LabelPad <pad>, {label}, <value>, ..., <value>`

Description: changes the label of the on-screen pad <pad>. Any number of strings, variables and values can be listed to create a label-string. There are 16 pads in Mozaic, numbered 0-15, but not all may be visible on every layout. This command is not to be confused with *LabelPads*, which changes the title above the pads section on the screen. Labels which are too long will be truncated on the screen (…).

Parameters and output:

`<pad>` The number of the pad to relabel. This number has to be in the range of 0-15

`{label}` text-string containing (part of) the new pad of the knob.

Example:

```
@OnLoad
  LabelPad 0, {Kick}
  LabelPad 1, {Snare}
@End
```

LabelPads

```
LabelPads {label}, <value>, ..., <value>
```

Description: Changes the title label above the pads section on the screen.

Parameters and output:

`{label}` text-string containing the new label of the title. If title is too long it will be truncated (…).

Example:

```
@OnNewBeat
  LabelPads {Drum Triggers}, num_active
@End
```

LabelXY

```
LabelXY {label}, <value>, ..., <value>
```

Description: Changes the title label above the XY pad on the screen.

Parameters and output:

`{label}` text-string containing the new label of the title. If title is too long it will be truncated (…).

Example:

```
@OnShiftUp
  LabelXY {Finger that Filter }, x * y
@End
```

LastAUParameter

```
<result> = LastAUParameter
```

Description: returns the number of the AU parameter which was most recently changed or updated by the AU host. It is recommended to only use this inside the *@OnAUParameter* event handler, or the result may be unpredictable.

Parameters and output:

`<result>` the last user AU parameter that was updated by the host. Result will be in the range of 0-7, as there are 8 parameters available for general purpose usage.

Example:

```
@OnAUParameter
  v = GetAUParameter LastAUParameter
  Log {The host changed parameter}, LastAUParameter, {to}, v
@End
```

LastKnob

```
<result> = LastKnob
```

Description: returns the number of the knob which was last tweaked. It is recommended to use this function only inside the *@OnKnobChange* event handler, as its value may be undefined in other events (due to multi-touch multiple knobs can be changed in parallel, which are all handled in separate event calls).

Parameters and output:

`<result>` the number of the knob which was last changed. Result can range from 0-21

Example:

```
@OnKnobChange
  Log (GetKnobValue LastKnob) // print the value of the last changed knob
@End
```

LastPad

```
<result> = LastPad
```

Description: returns the number of the most recent pad which was pushed or released. It is recommended to use this function only inside the *@OnPadDown* or *@OnPadUp* event handlers, as its value may be undefined in other events (due to multi-touch).

Parameters and output:

`<result>` the last pad that was touched or released on the screen. Result ranges 0-15.

Example:

```
@OnPadDown
  Log {You tapped pad }, LastPad
@End
```

LastPadVelocity

```
<result> = LastPadVelocity
```

Description: returns the velocity of the last user interaction with a pad on the screen. The velocity value is derived from the distance to the center of the pad. If you tap on the center, velocity will be 127, and this will be lower the further away from the center you tap.

Parameters and output:

`<result>` the velocity of the tap on the last pad that was touched on the screen.

Example:

```
@OnPadDown
  SendMIDINoteOn 0, 36, LastPadVelocity
  SendMIDINoteOff 0, 36, 0, 500.0
@End
```

LatchPad

`LatchPad <pad>, <state>`

Description: Switches the background LED for Pad <pad> on or off.

Parameters and output:

`<pad>` the number of the pad you want to control. Range can be 0-15.

`<state>` the desired state of the pad: YES/true or NO/false

Example:

```
@OnMidiNoteOn
  LatchPad 0, YES
@End

@OnMidiNoteOff
  LatchPad 0, NO
@End
```

Log

`Log <value>, <value>, {string}, ..., <value>`

Description: Prints any number of values and strings to the Log window. Put as many things in here as you like, as long as each element is separated by a comma. This is a great tool for passing runtime information about your script to the user or for checking values while debugging your script.

Parameters and output:

`<value>` any value, variable or function call (function calls with parameters should be put inside brackets).

`{string}` any text string.

Example:

```
Log {The current tempo is }, HostTempo
Log (Random 0, 10), { is a number between 0 and 10.}
Log var0, var1, {some text}, somearray[0]
```

Log10

```
<result> = Log10 <value>
```

Description: returns the (base 10) logarithm for <value>, as long as <value> is not negative, or zero.

Parameters and output:

`<result>` the base-10 logarithm for <value>

`<value>` input value or variable

Example:

```
v = Log10 v // get the logarithm of the input value
```

Logn

```
<result> = Logn <value>
```

Description: returns the natural logarithm for <value>, as long as <value> is not negative, or zero.

Parameters and output:

`<result>` the natural logarithm for <value>

`<value>` input value or variable

Example:

```
v = Logn v// get the natural logarithm of the input value
```

LogTime

```
LogTime
```

Description: prints information about the current event handler, the metronome and host timing on the Log screen. This is mostly a debugging function, letting you know some important system states while running your script.

Example:

```
@OnMidiInput
  LogTime
@End
```

MIDIByte1, MIDIByte2, MIDIByte3

```
<result> = MIDIByte1
<result> = MIDIByte2
<result> = MIDIByte3
```

Description: return the raw data of the incoming MIDI messages. Regular (non-sysex) MIDI messages can either 1, 2 or 3 bytes long. Check the event type of the incoming MIDI (using *MIDICommand*) to make sure these bytes contain meaningful data.

Parameters and output:

`<result>` the raw value (ranging 0-255) of the incoming MIDI event data

Example:

```
@OnMidiNote
  vel = MIDIByte3 // the third byte contains velocity in note messages
  if vel = MIDIVelocity
    Log {That was to be expected}
  endif
@End
```

MIDIChannel

`<result> = MIDIChannel`

Description: this will tell you the MIDI channel number (0-15) of the incoming MIDI message. MIDIChannel and MIDICommand added together form the first byte of the MIDI event.

Parameters and output:

`<result>` MIDI channel number of the incoming MIDI event. Set to 0 if the MIDI event is a global event (i.e. an event which is not channel specific).

Example:

```
@OnMidiNote
  SendMIDIOut (LastMIDICommand + LastMIDIChannel), MIDIByte2, MIDIByte3
  Log {The MIDI Channel is: }, LastMIDIChannel + 1
@End
```

MIDICommand

`<result> = MIDICommand`

Description: returns the current MIDI command. If it is a channel-specific command (such as a note message or a CC event) the channel information will be removed (and stored separately in *MIDIChannel*).

So; a MIDI Note On message would be reported as 0x90, a MIDI CC message would become 0xB0, etc. Simply add the value in MIDIChannel to recreate the original MIDI message.

Global messages (such as clock events) don’t have channel information and are reported exactly as they were received.

Parameters and output:

`<result>` MIDI command of the event that is currently being handled.

Example:

```
@OnMidiInput
  if MIDICommand = 0x90
    Log {we received a NOTE ON message}
  elseif MIDICommand = 0x80
    Log {we received a NOTE OFF message}
  else
    Log {we received the following MIDI Command: }, MIDICommand
  endif
@End
```

MIDINote

`<result> = MIDINote`

Description: returns the current MIDI note value, range 0-127. Only use this in *@OnMIDINote*, *@OnMIDINoteOn*, *@OnMIDINoteOff,* or the value may be undefined. This is mostly a convenience function, because for note on and note off events *MIDINote* will always have the same value as *MIDIByte2* (and both can be used interchangeably).

Parameters and output:

`<result>` Note number of the note event currently being handled.

Example:

```
@OnMidiNote
  if MIDINote < 36
    SendMIDIOut MIDICommand + MIDIChannel, MIDINote + 12, MIDIVelocity
  else
    SendMIDIThru
  endif
@End
```

MIDISustainPedalDown

`<result> = MIDISustainPedalDown`

Description: returns the state of the sustainpedal(s). If any sustain pedal is currently activated, this function will return YES (true). Otherwise it will return NO (false).

Parameters and output:

`<result>` The state of the sustainpedal(s).

Example:

```
@OnMidiNote
  if MIDISustainPedalDown
    SendMIDIThruOnCh 2
  else
    SendMIDIThruOnCh 0
  endif
@End
```

MIDIVelocity

```
<result> = MIDIVelocity
```

Description: returns the velocity of the note you're currently handling. It is recommended to only use this function inside *@OnMIDIxxxx* handler events.

Parameters and output:

`<result>` Note velocity of the note event currently being handled.

Example:

```
@OnMidiNote
  newvel = MIDIVelocity * 1.33
  newvel = Clip newvel, 0, 127
  SendMIDIOut MIDIByte1, MIDIByte2, newvel
@End
```

MotionPitch, MotionRoll, MotionYaw

```
<result> = MotionPitch
<result> = MotionRoll
<result> = MotionYaw
```

Description: these three functions return the current state of the tilt-sensors, using the built-in motion co-processors of the iOS device. Each function returns the angle of one of the 3 axes:

- MotionPitch: the angle which changes when your device makes a looping
- MotionRoll: changes when your device makes a barrelroll
- MotionYaw: the angle movement of a compass

Each of these values is scaled from 0-127 (full rotation) compared to the reference angle. The reference angle is the way you hold the device the moment you load the script. This is the center-value for each axis: 64, the reference angle.

Parameters and output:

`<result>` Current angle of the X, Y or Z axis

Example:

```
@OnMetroPulse
  SendMIDICC 0, 13, MotionPitch
  SendMIDICC 0, 15, MotionRoll
  SendMIDICC 0, 17, MotionYaw
@End
```

NoteName

```
{notename} = NoteName <note> [,<showoctave>]
```

Description: Returns a string containing the name of the numerical note value passed in <note>. For example: 0 = C, 1 = C#, etc. Optionally an octave number can be appended (D#4) if <showoctave> is set to YES/true.

This function is a string macro which can only be used inside *Log* or *Label* functions. Using it outside one of these functions will result in an error message.

Parameters and output:

`{notename}` Name of the note indicated by <note>

`<note>` Number of the note you want to have converted to a string

`<showoctave>` Boolean value to indicate if an octave number must be appended to the string.

Example:

```
@OnMidiNote
  Log {Note: }, (NoteName MIDINote, YES), { detected}
@End
```

PadState

```
<result> = PadState <padnum>
```

Description: Returns the current latch-state of pad <padnum>. Latch state means whether the backlight LED of a pad is continuously on. You can change the state using *LatchPad*.

Parameters and output:

`<result>` Boolean value indicating the current latch state. YES/true = on, NO/false = off.

`<padnum>` Number of the pad you want to check (0-15)

Example:

```
@OnNewBeat
  state = PadState 0
  LatchPad 0, NOT state // blinky blinky!
@End
```

Pow

```
<result> = Pow <base>, <exponent>
```

Description: Computes the value of base raised to the power exponent.

Parameters and output:

`<result>` The value of base raised to the power exponent

`<base>` The base value

`<exponent>` Exponent for the power calculation

Example:

```
var0 = Pow 100, 0.8
Log var0
```

PresetScale

```
PresetScale {scalename}
PresetScale <scalenumber>
```

Description: Activates one of the preset scales. If you want to define your own custom scale, you can use the *CustomScale* command.

Parameters and output:

`{scalename}` String containing the name of the preset scale you wan to activate.

`<scalenum>` Number of the preset scale you wan to activate.

Available scales:

```
0 {Chromatic},
1 {Major},
2 {Minor},
3 {MinorMelodic},
4 {MinorHarmonic},
5 {MajorPentatonic},
6 {MinorPentatonic},
7 {Aeolian},
8 {Dorian},
9 {Lydian},
10 {Mixolydian},
11 {Phrygian},
12 {Blues},
13 {WholeTone},
14 {Diminished},
15 {Bhairavi},
16 {Gypsy},
17 {Klezmer},
18 {Octave},
19 {Andean},
20 {Iwato},
21 {InSen},
22 {HiraJoshi},
23 {Pelog},
24 {Yo}
```

Example:

```
PresetScale {MinorPentatonic}  // same as: PresetScale 6
SetRootNote 0 // set root note to C
nn = ScaleQuantize MIDINote // quantize the note to Minor Pentatonic scale
```

QuarterNote

```
<result> = QuarterNote
```

Description: Returns the duration (in milliseconds) of a quarter note, given the current tempo. A quarter note is the same as 1 beat.

Parameters and output:

`<result>` The duration of a quarter note in milliseconds

Example:

```
SendMIDICC 0, 7, 64, QuarterNote * 2 // send CC after 0.5 measure
```

Random

```
<result> = Random <min>, <max>
```

Description: Generates a random value in the range <min> to <max>. Values are generated as integer numbers without fractions.

Parameters and output:

`<result>` A random value in the range of <min> to <max>

`<min>` Lower end of the range.

`<max>` Upper end of the range

Example:

```
rnd = Random 0, 100 // a number between 0 - 100
rnd = rnd / 100    // var0 is now a random number between 0.0 and 1.0
```

ReceiveSysex

`ReceiveSysex <array>`

Description: When called from an *OnSysex* event handler, this command will copy the incoming Sysex data into a variable. The data will never exceed the 1024 byte limit that can be held inside a regular Mozaic array variable.

The actual size of the received data (i.e. the number of array elements) can be queried using the *SysexSize* function.

Parameters and output:

`<array>` A variable for containing the Sysex data. If the variable does not yet exist it will be created.

Example:

```
@OnSysex
  ReceiveSysex sysexdata
  sysexdatasize = SysexSize
@End
```

ResetLFO

`ResetLFO <lfo> [,<startphase>]`

Description: Resets the LFO <lfo> to its starting phase, or optionally to a specific phase.

Parameters and output:

`<lfo>` The number of the LFO that needs to be reset (range 0-15)

`<startphase>` Optional starting phase for the LFO's period:
0.0 = starting phase
0.5 = midpoint of period
1.0 = end of period
In a standard Sine LFO 0.25 is the highest point of the period, 0.75 is the lowest point.

Example:

```
@OnHostStart
  for lfonum = 0 to 15
    ResetLFO lfonum // restart all LFOs
  endfor
@End
```

ResetNoteStates

`ResetNoteStates [<default>]`

Description: Resets the NoteState matrix. Refer to *SetNoteState* or *GetNoteState* for more details.

Parameters and output:

`<default>` An optional value used to fill all cells. If not specified all cells will be filled with 0.

Example:

```
@OnShiftDown
  // this will erase the NoteState matrix and fill all cells with -1
  ResetNoteStates -1
@End
```

ResetTimer

`ResetTimer`

Description: Resets the timer countdown to the full interval time. So if the timer's interval time was set to one minute, the timer event will be triggered one minute after *ResetTimer* was called. If *ResetTimer* is called before that time, the timer event will be postponed again.

Example:

```
@OnMidiNote
  ResetTimer
@End

@OnTimer
  Log {You haven't played a note in a while!}
@End
```

RootNoteName

```
{notename} = RootNoteName
```

Description: Returns a string containing the name of the root note of the currently active scale.

This function is a string macro which can only be used inside *Log* or *Label* functions. Using it outside one of these functions will result in an error message.

Parameters and output:

`{notename}` Name of the root note of the current scale

Example:

```
@OnMidiNote
  Log {Current scale: }, ScaleName, { Current Root note: }, RootNoteName
@End
```

Round, RoundUp, RoundDown

```
<result> = Round <value>
<result> = RoundUp <value>
<result> = RoundDown <value>
```

Description: All three functions round the incoming value to an integral value. *Round* always rounds to the nearest integral value. *RoundUp* always rounds away from zero, and *RoundDown* always rounds towards zero.

Parameters and output:

`<result>` The value rounded to an integral value

`<value>` The input value

Example:

```
v = 3.45
Log (Round v)       // 3
Log (RoundUp v)     // 4
Log (RoundDown v)   // 3
```

ScaleName

```
{scalename} = ScaleName [<scalenum>]
```

Description: Returns a string containing the name of the scale indicated by <scalenum>, or the currently active scale if <scalenum> is not specified.

This function is a string macro which can only be used inside *Log* or *Label* functions. Using it outside one of these functions will result in an error message.

Parameters and output:

`{scalename}` Name of the current scale

`<scalenum>` Optional number of the scale you want to retrieve the name of. If left out the name of the currently active scale will be returned.

Example:

```
@OnMidiNote
  Log {Current scale: }, ScaleName, { Current Root note: }, RootNoteName
@End
```

ScaleQuantize

`<result> = ScaleQuantize <midinote>`

Description: takes the note <midinote>, checks if it matches the currently active musical scale (set with *PresetScale* or *CustomScale*), respecting the current root note (set with *SetRootNote*). If there is a match, the same note is returned. If it doesn't match the active scale, it returns the next higher note which does.

Parameters and output:

`<result>` A midi note number (range 0-127) which matches the active scale

`<midinote>` A midi note number (range 0-127)

Example:

```
@OnMIDINote
  nn = ScaleQuantize MIDINote // quantize the incoming midi note
  SendMIDIOut MIDIByte1, nn, MIDIByte3 // send the MIDI note out
@End
```

SendMIDIBankSelect

`SendMIDIBankSelect <chan>, <value> [,<delay>]`

Description: Sends a MIDI Bank Select message

Parameters and output:

`<chan>` The desired MIDI channel (0-15)

`<value>` Number of the bank

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnLoad
  SendMIDIBankSelect 0, 1
  SendMIDIProgramChange 0, 24 // select patch 24 in bank 1
@End
```

SendMIDICC

`SendMIDICC <chan>, <controller>, <value> [,<delay>]`

Description: Sends a MIDI CC message

Parameters and output:

`<chan>` The desired MIDI channel (0-15)

`<controller>` The CC number you want to change (0-127)

`<value>` The value which needs to be sent to the Continuous Controller (0-127)

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnMetroPulse
  v = GetLFOValue 0     // get the current value of LFO 0
  SendMIDICC 0, 15, v   // send the value to MIDI CC 15
@End
```

SendMIDINoteOff

`SendMIDINoteOff <chan>, <note>, <velocity> [,<delay>]`

Description: Sends a MIDI Note Off message (0x80).

Parameters and output:

`<chan>` The desired MIDI channel (0-15)

`<note>` The MIDI note number (0-127)

`<velocity>` The velocity of the note

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnShiftDown
  // do a MIDI panic on channel 0
  for nn = 0 to 127
    SendMIDINoteOff 0, nn, 0
  endfor
@End
```

SendMIDINoteOn

`SendMIDINoteOn <chan>, <note>, <velocity> [,<delay>]`

Description: Sends a MIDI Note On message (0x90).

Parameters and output:

`<chan>` The desired MIDI channel (0-15)

`<note>` The MIDI note number (0-127)

`<velocity>` The velocity of the note

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnShiftDown
  // send a chord
  SendMIDINoteOn 0, 36, 100
  SendMIDINoteOn 0, 40, 100
  SendMIDINoteOn 0, 43, 100
@End

@OnShiftUp
  // send stop it
  SendMIDINoteOff 0, 36, 0
  SendMIDINoteOff 0, 40, 0
  SendMIDINoteOff 0, 43, 0
@End
```

SendMIDIOut

`SendMIDIOut <byte1>, <byte2>, <bute3> [,<delay>]`

Description: Sends out a free-form MIDI message defined by the contents of <byte1>, <byte2> and <byte3>. Some MIDI messages require fewer than three bytes, but Mozaic checks this for you and sends out the correct number of bytes accordingly.

Parameters and output:

`<byte1>` MIDI event message

`<byte2>` first data byte, contents depend on the event message

`<byte3>` second data byte, contents depend on the event message

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnMIDIInput
  SendMIDIOut MIDIByte1, MIDIByte2, MIDIByte3 // forward all incoming MIDI
@End
```

SendMIDIPitchbend

`SendMIDIPitchbend <chan>, <value> [,<delay>]`

Description: Sends a MIDI pitchbend event. <value> is the 14-bit pitchbend value, with a range of 0-16383. The center value (no bend) is 8192 in standard MIDI.

Parameters and output:

`<chan>` The desired MIDI channel (0-15)

`<value>` 14 bit pitchbend value (0-16383)

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnXYChange
   y = GetYValue * 128
   SendMIDIPitchbend 0, y
@End
```

SendMIDIProgramChange

`SendMIDIProgramChange <chan>, <patch> [,<delay>]`

Description: Sends a MIDI Program Change (PC) message

Parameters and output:

`<chan>` The desired MIDI channel (0-15)

`<patch>` Number of the patch

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnLoad
  SendMIDIBankSelect 0, 1
  SendMIDIProgramChange 0, 24 // select patch 24 in bank 1
@End
```

SendSysex

`SendSysex <array>, <length> [,<checksum>, <startindex>]`

Description: Ouputs the contents of an array variable in a Sysex message. Sysex start/end-bytes will be added automatically, so these don't need to be part of the message and the length calculation.

Optionally a checksum can be performed. The results of this checksum will replace the last byte of your message, so make sure you add a 'dummy' byte at the end of your message if you want to use the checksum feature.

Parameters and output:

`<array>` The variable containing the Sysex data to be sent

`<length>` The number of bytes/elements in the array that need to be sent (including the optional checksum byte)

`<checksum>` Optional checksum calculation to be performed:
0 - No checksum (default)
1 - Roland/Boss checksum
2 - Fractal Audio checksum

`<startindex>` Optional start index for the checksum calculation. For Roland/Boss checksums this is usually the start of the parameter address section of the Sysex. This parameter has no effect on other checksum calculations.

Example:

```
@OnTimer
  data = [0x13, 0x37, 0xB0, 0x55, value]
  SendSysex data, 5
@End
```

SendSysexThru

`SendSysexThru`

Description: Forwards the last received Sysex message without any changes.

Example:

```
@OnSysex
  SendSysexThru // make sure our script lets Sysex go through
@End
```

SendMIDIThru, SendMIDIThruOnCh

`SendMIDIThru [<delay>]`
`SendMIDIThruOnCh <chan> [,<delay>]`

Description: Forwards incoming MIDI events to the output port. Mozaic doesn't automatically do this (because it's not always wanted or needed), so you'll have to manually make sure MIDI is sent through as needed. These are convenience functions to do that for you.

SendMIDIThru forwards any incoming MIDI indiscriminately. SendMIDIThruOnCh does the same thing, but sends everything to a specific MIDI channel, regardless of what channel the incoming MIDI message was sent on.

> Note that if you're handling 'overlapping events' (such as *OnMIDIInput*, *OnMIDINote* and *OnMIDINoteOn*) you have to be careful that you're only forwarding incoming events once.

Parameters and output:

`<chan>` The desired MIDI channel (0-15)

`<delay>` Optional delay (in milliseconds) before the MIDI message is sent out. If left out or set to 0, the message is sent out immediately.

Example:

```
@OnMIDIInput
  if ShiftPressed = NO
    SendMIDIThru
  else
    SendMIDIThruOnCh 6 //only when the shift button is pressed!
  endif
@End
```

SetAUParameter

`SetAUParameter <parameter>, <value>`

Description: Sets one of the user-definable AU parameters <parameter> to value <value>.

Parameters and output:

`<parameter>` The number of the AU parameter (0-7)

`<value>` Value the parameter needs to be set to (0-127). This can be any 32 bit floating point value, and doesn't need to be a 7 bit integer value like MIDI CC values.

Example:

```
@OnKnobChange
  if LastKnob = 2
    v = GetKnobValue 2
    SetAUParameter 0, v // send the value of knob 2 to parameter 0
  endif
@End
```

SetKnobValue

`SetKnobValue <knob>, <value>`

Description: Sets the value of knob number <knob> to new value <value>. This will also change the setting of the knob on the screen, but will not trigger an *@OnKnobChange* event.

Parameters and output:

`<knob>` The number of the knob to be changed (0-21)

`<value>` New value of the knob (0-127). Doesn't have to be an integral value, can be floating point.

Example:

```
@OnXYChange
  x = GetXValue
  y = GetYValue
  SetKnobValue 0, x
  SetKnobValue 1, y
@End
```

SetLFOType

`SetLFOType <lfo>, {wave}`

Description: Selects the waveform for LFO number <lfo>. Changing the waveform will not reset the current phase of the waveform and may result in discontinuities.

Parameters and output:

`<lfo>` The number of the LFO (0-15)

`{wave}` String containing the name of the waveform you want to select

Available waveforms:

`{Sine}, {Cosine}, {Square}, {Triangle}, {RampUp}, {RampDown}, {SH}`

Example:

```
// set all LFOs to Sample & Hold:
for lfonum = 0 to 15
  SetLFOType lfonum, {SH}
endfor
```

SetMetroPPQN

`SetMetroPPQN <ppqn>`

Description: The the number of pulses per quarter note (i.e. events per beat) for the programmable metronome. Every pulse generates an @OnMetroPulse event. Pulse events are always synced to the host tempo. Higher values result in more pulses being sent (i.e. smaller intervals) but may also lead to higher CPU load, because events need to be handled more often. You can optionally apply a swing/shuffle feel to the metronome using *SetMetroSwing*.

Parameters and output:

`<ppqn>` The number of pulse events generated per quarter note. Range is 1-384

Example:

```
@OnLoad
  SetMetroPPQN 3 // triplet feel
  SetMetroSwing 5 // add a subtle 5% swing to the metronome
@End
```

SetMetroSwing

`SetMetroSwing <percentage>`

Description: Adds an amount of shuffle feel to the metronome. Useful if you're building your own sequencer or generative program. The level of swing is defined as a percentage.

Parameters and output:

`<percentage>` The amound of swing you want to apply to the metronome (0-100).

Example:

```
@OnLoad
  SetMetroPPQN 3 // triplet feel
  SetMetroSwing 5 // add a subtle 5% swing to the metronome
@End
```

SetNoteState

`SetNoteState <chan>, <note>, <value>`

Description: Sets a value we want to remember for note <note> in channel <chan>. The NoteState matrix is a 2-dimensional array, a virtual 'locker room', with a cell for each possible note which lets you store an arbitrary value and retrieve it when needed.

For example when you manipulate an incoming note in a NoteOn handler, you can then save some useful information in the note state matrix and retrieve that when the NoteOff event for that note is triggered.

The value can be numerical or boolean.

Parameters and output:

`<chan>` The channel number of the note (0-15)

`<note>` The note number (0-127)

`<value>` The value we want to remember. This can be numerical or boolean.

Example:

```
@OnMidiNoteOn
  // store a random number for this note
  SetNoteState MIDIChannel, MIDINote, (Random 0, 5)
@End

@OnMidiNoteOff
  // recall the value we stored
  v = GetNoteState MIDIChannel, MIDINote
  Log {We stored: }, v
@End
```

SetRootNote

`SetRootNote <notenum>`

Description: specifies the root note of the currently active musical scale. Has no effect when {Chromatic} is the active scale.

Parameters and output:

`<notenum>` The number of the active root note, where C is defined as 0, the default root note. So: C = 0, C# = 1, D = 2, D# = 3, E = 4, F = 5, F# = 6, G = 7, G# = 8, A = 9, A# = 10, B = 11

Example:

```
@OnLoad
  PresetScale {Major}
  SetRootNote 5 // F
@End
```

SetShortName

`SetShortName {shortname}`

Description: sets the “short name” for the plugin instance. If supported by the AU host, it may help distinguish between what different instances of Mozaic plugins do.

Parameters and output:

`{shortname}` The Audio Unit “short name” for this plugin instance. If supported by AU hosts, it can typically be 5-8 characters long and will be truncated if longer.

Example:

```
@OnLoad
  SetShortName {SlowLFO}
@End
```

SetTimerInterval

`SetTimerInterval <ms>`

Description: Sets the interval between timer events. Timer events are completely independent of the AU host's tempo, and do not rely on the host being in playback mode.

Parameters and output:

`<ms>` The interval between timer events, specified in milliseconds. Timer events are always generated with sample-accuracy so they can be used for very tight timing purposes, such as generating rhythmic MIDI output.

Example:

```
@OnLoad
  SetTimerInterval 250.0 // 4 x per second
  StartTimer // begin sending timer events
@End
```

SetupLFO

`SetupLFO <lfo>, <min>, <max>, <sync>, <freq>`

Description: Sets up the LFO with number <lfo>, giving it an output range of <min> to <max>. If <sync> is set to YES/true it will sync to the tempo and do <freq> periods per measure. If <sync> is false/NO, the LFO will not be synced to the host's tempo and <freq> will be the LFO wave's period length in Hz. (i.e. the number of periods per second).

The default LFO waveform is a sinewave. If you wan to change the LFO waveform you can use *SetLFOType*.

Parameters and output:

`<lfo>` The number of the LFO which needs to be started (0-15)

`<min>` The lower range of the LFO: the value of the lowest point of the LFO.

`<max>` The upper range of the LFO: the value of the highest point of the LFO.

`<sync>` Boolean value, indicating whether the LFO should be synced to the AU host's tempo.

`<freq>` If synced to tempo this is the number of periods per measure. If not synced to tempo this is the number of periods per second.

Example:

```
@OnLoad
  SetLFOType 0, {SH} // sample and hold
  SetupLFO 0, 0, 127, NO, 0.5
@End
```

SetXYValues

`SetXYValues <xvalue>, <yvalue>`

Description: Updates the values of the XY pad with new values <xvalue> and <yvalue>. Also updates the pad on the screen, but does not generate an *@OnXYChange* event.

Parameters and output:

`<xvalue>` the new value for the x-axis (0-127)

`<yvalue>` the new value for the y-axis (0-127)

Example:

```
@OnLoad
  var0 = random 0, 127
  var1 = random 0, 127
  SetXYValues var0, var1
@End
```

ShiftPressed

`<result> = ShiftPressed`

Description: returns the state of the on-screen Shift button. If the button is pressed, this function returns YES (true). If not, it will return NO (false).

Parameters and output:

`<result>` The state of the Shift button

Example:

```
@OnTimer
  if ShiftPressed = true // true = YES = 1, you can use whatever you prefer
    SetKnobValue 0, (Random 0,127)
  endif
@End
```

ShowLayout

`ShowLayout <layout>`

Description: Changes the on-screen GUI to a different layout, numbered <layout>. It is recommended to do this @*OnLoad*.

Parameters and output:

`<layout>` The number of the preferred layout (0-4). There are five different layouts available:
0: Mix: 4 trigger pads, 10 knobs and an XY pad
1: Knobs: 22 knobs
2: Pads: 16 trigger pads and 4 knobs
3: Sliders: 10 sliders and an XY Pad (sliders are treated in scripts as if they are knobs)
4: Minimal: 4 knobs and a description text box

Example:

```
@OnLoad
  ShowLayout 2 // show the view with 16 pads
@End
```

Sin

`<result> = Sin <value>`

Description: Computes the sine of <value>

Parameters and output:

`<result>` The sine of <value>

`<value>` The input value

Example:

```
v = Sin (phase * 2 * 3.14159)
```

Sqrt

`<result> = Sqrt <value>`

Description: Computes the square root of <value>

Parameters and output:

`<result>` The square root of <value>

`<value>` The input value

Example:

```
pyt = Sqrt ((delta1 * delta1) + (delta2 * delta2)) // pythagoras
```

StopTimer

`StopTimer`

Description: Stops the timer. No more *@OnTimer* events will be generated until the timer is restarted using *StartTimer*.

Example:

```
@OnHostStart
  StartTimer
@End

@OnHostStop
  StopTimer
@End
```

SysexSize

`<result> = SysexSize`

Description: returns the size of the last received Sysex message

Parameters and output:

`<result>` the size of the last received Sysex message

Example:

```
@OnSysex
  ReceiveSysex data
  ssize = SysexSize
@End
```

SystemTime

`<result> = SystemTime`

Description: returns the current system time in milliseconds. Useful to measuring the duration of events and processes.

Parameters and output:

`<result>` the current iOS system time in millisecond ticks

Example:

```
@MyTimeDelta
  newtime = SystemTime
  Log (newtime - lasttime)
  lasttime = newtime
@End
```

Tan

`<result> = Tan <value>`

Description: Returns the tangent of <value>

Parameters and output:

`<result>` The tangent of <value>

`<value>` The input value

Example:

```
v = Tan (x / y)
```

Tanh

`<result> = Tanh <value>`

Description: Returns the hyperbolic tangent of <value>

Parameters and output:

`<result>` The hyperbolic tangent of <value>

`<value>` The input value

Example:

```
v = Tanh (x * y)
```

TranslateCurve

`TranslateCurve <input>, <curve>, <min>, <max>`

Description: Applies a non-linear power curve to an input value, to add a bias for lower or higher values. <min> and <max> apply to both the input and the output range.

You can use this, for example, to add a bias to note velocity values.
- A curve of 1.0 means there is a linear translation, in other words: the output will be the same as the input
- If the curve is in the range of 0.0-1.0, there will be a bias towards higher values
- If the curve is > 1.0 the bias will be towards lower values

Parameters and output:

`<input>` The input value. Should be in the range of <min> to <max>

`<curve>` The curve of the bias. 1 = linear, 0-1 is bias on higher values, >1 bias on lower values

`<min>` The lower range of the input/output scale

`<max>` The upper range of the input/output scale

Example:

```
@OnMidiNote
  vel = TranslateCurve MIDIVelocity, 0.3, 0, 127 // amplify velocity
  SendMIDIOut MIDIByte1, MIDIByte2, vel
@End

@OnKnobChange
  val = GetKnobValue LastKnob
  val = TranslateCurve val, 2.0, 0, 127 // emphasize lower range of scale
  Log val
@End
```

TranslateScale

`TranslateScale <input>, <inputmin>, <inputmax>, <outputmin>, <outputmax>`

Description: Translates a value from one range/scale to another. You simply specify the input range (which the input value falls into) and the output range. E.g. you can use this to compute how LFO values translate to a 0-127 MIDI CC value etc.

This does the essentially the same as:
*output = (((input - inputmin) / inputrange) * outputrange) + outputmin*

Parameters and output:

`<input>` The input value. Should be in the range of <inputmin> to <inputmax>

`<inputmin>` The lower end of the input range

`<inputmax>` The upper end of the input range

`<outputmin>` The lower end of the output range

`<outputmax>` The upper end of the output range

Example:

```
SetupLFO 0, -50, 50, NO, 0.5
v = GetLFOValue 0                 // v is in range -50 to 50 now
v = TranslateScale v, -50, 50, 0, 127 // translate v to MIDI CC range 0 to 127
```

Unassigned

`<bool> = Unassigned <var>`

Description: Tests whether variable <var> exists and has been assigned a valid value. This can be used to make sure that variables are initialized when first created, and prevent overwriting any previously loaded values (e.g. from state saving/restoring).

Parameters and output:

`<bool>` YES/true when the variable *does not* exist yet, NO/false when the variable already exists and has valid value assigned to it.

`<var>` The variable we want to test for.

Example:

```
@OnLoad
  if Unassigned testvariable
    // testvariable does not exist yet, initialize it now!
    testvariable = 0
  endif
@End
```

10. Mozaic Function overview per category

A handy cheat sheet for quickly looking up related functions and commands.
This list is also available in the app.

MIDI functions

```
SendMIDIOut <byte1>, <byte2>, <byte3> [,<delay_in_milliseconds>]
SendMIDINoteOn <chan>, <note>, <velocity> [,<delay_in_milliseconds>]
SendMIDINoteOff <chan>, <note>, <velocity> [,<delay_in_milliseconds>]
SendMIDICC <chan>, <controller>, <value> [,<delay_in_milliseconds>]
SendMIDIPitchbend <chan>, <value> [,<delay_in_milliseconds>]
SendMIDIProgramChange <chan>, <value> [,<delay_in_milliseconds>]
SendMIDIBankSelect <chan>, <MSB>, <LSB> [,<delay_in_milliseconds>]
SendMIDIThru [<delay_in_milliseconds>]
SendMIDIThruOnCh <chan> [,<delay_in_milliseconds>]
ConfigureMPE <lowerzone>, <upperzone>
<var> = MIDIChannel
<var> = MIDICommand
<var> = MIDINote
<var> = MIDIVelocity
<var> = MIDIByte1
<var> = MIDIByte2
<var> = MIDIByte3
<bool> = MIDISustainPedalDown
```

Sysex functions

```
SendSysex <array>, <length> [,<checksum>, <checksum start index>]
SendSysexThru
ReceiveSysex <array>
<var> = SysexSize
```

AUv3 and Host functions

```
SetShortName {name}
SetMetroPPQN <ppqn>
SetMetroSwing <swing%>
SetAUParameter <parameter>, <value>
<var> = HostTempo
<var> = HostBar
<var> = HostBeat
<var> = HostBeatsPerMeasure
<bool> = HostRunning
<var> = CurrentMetroPulse
<var> = GetAUParameter <parameter>
```

```
<var> = LastAUParameter
<ms> = QuarterNote
```

Timer and LFO functions

```
StartTimer
StopTimer
ResetTimer
SetTimerInterval <milliseconds>
SetupLFO <lfo>, <minimum>, <maximum>, <sync>, <frequency>
SetLFOType <lfo>, {type}
ResetLFO <lfo> [,<phase>]
<var> = GetLFOValue <lfo>
```

Available LFO types: Sine, Cosine, Square, Triangle, RampUp, RampDown, SH

Musical Scale functions

```
CustomScale c, c# d, d#, e, f, f#, g, g#, a, a#, b
PresetScale {scale}
PresetScale <scalenumber>
SetRootNote <rootnote>
<bool> = InScale <note>
<var> = ScaleQuantize <note>
```

Available preset scales: Chromatic (0), Major (1), Minor (2), MinorMelodic (3), MinorHarmonic (4), MajorPentatonic (5), MinorPentatonic (6), Aeolian (7), Dorian (8), Lydian (9), Mixolydian (10), Phrygian (11), Blues (12), WholeTone (13), Diminished (14), Bhairavi (15), Gypsy (16), Klezmer (17), Octave (18), Andean (19), Iwato (20), InSen (21), HiraJoshi (22), Pelog (23), Yo (24)

GUI and Interaction functions

```
Exit
ShowLayout <layout>
LabelPad <pad>, {label}
LabelPads {title}
LabelKnobs {title}
LabelXY {title}
LabelKnob <knob>, {label}
SetKnobValue <knob>, <value>
ColorPad <padnum>, <colornum>
LatchPad <button>, <state>
FlashPad <button>
FlashUserLed
SetXYValues <x>, <y>
Log <var>, {text}, ...
LogTime
```

```
<var> = GetXValue
<var> = GetYValue
<var> = GetXYMorphValue <topleft>, <topright>, <bottomleft>, <bottomright>
<var> = LastKnob
<var> = GetKnobValue <knob>
<var> = LastPad
<var> = LastPadVelocity
<var> = PadState <padnum>
<var> = MotionPitch
<var> = MotionRoll
<var> = MotionYaw
<bool> = ShiftPressed
{notename} = NoteName <note> [,<showoctave>]
{notename} = RootNoteName
{scalename} = ScaleName [<scalenum>]
```

Variables and Math functions

```
FillArray <var>, <value>
CopyArray <sourcevar>, <destvar> [, <numcells>]
Inc <var> [,<max>]
Dec <var> [,<min>]
<bool> = Unassigned <var>
<var> = Round <value>
<var> = RoundUp <value>
<var> = RoundDown <value>
<var> = Random [<min>, <max>]
<var> = Clip <var>, <min>, <max>
<var> = TranslateCurve <value>, <curve> [,<min>, <max>]
<var> = TranslateScale <value>, <inputmin>, <inputmax>, <outputmin>, <outputmax>
<var> = Sin <value>
<var> = Cos <value>
<var> = Tan <value>
<var> = Tanh <value>
<var> = Exp <value>
<var> = Sqrt <value>
<var> = Abs <value>
<var> = Logn <value>
<var> = Log10 <value>
<var> = Pow <base>, <exponent>
<int> = Div <int>, <int>
```

NoteState matrix functions

```
ResetNoteStates [<defaultvalue>]
SetNoteState <channel>, <note>, <var>
<var> = GetNoteState <channel>, <note>
```