Introduction to scripting with Gig Performer

Draft V0.51

Dr. David H Jameson
Nebojsa Djogo

Deskew Technologies, LLC

August 5, 2018

Introduction

The Gig Performer scripting language, which we have rather unimaginatively named **GP Script**, is a special purpose language designed to add programmable and responsive functionality to Gig Performer. For example, you can write scripts with code that can respond to incoming MIDI note events and then transpose them, make chords out of them constrained to a scale, or use them to change widget values or to directly control parameters of a plugin (keyboard tracking, anyone?). Script code can be triggered in response to your moving a knob, slider or button. There are also time generators that can be used as LFOs (with varying shapes) or as ADSRs.

Events and callbacks

Generally, GP Script is event driven. It doesn't do anything until something "happens" such as your playing a key, turning a knob, switching to another rackspace or to a different variation. A certain amount of time passing can also count as something "happening". All of these examples are known as events and when you write code that can respond to an event, that code is called a callback. GP Script can currently handle the following events:

- Initialization
- Rack activation
- Rack deactivation
- Variation change
- Midi events (notes, CC values, aftertouch, etc.)
- Widget changes
- Plugin parameter changes
- LFOs (also known as Time Generators)
- OSC Messages (coming soon)

A simple example

```
var    pi : double
initialization
   pi = 3.14159
   Print(pi)
end
```

So, what's going on here? Well, first we declared a variable called pi and whose value must be a floating-point number (double). In GPScript all variables must be declared, and they must have a **type**. The **type** of a variable defines the kinds of values that the variable can represent and also determines what kinds of operations can be applied to it. So called strongly typed languages make it easier to prevent certain kinds of common bugs.

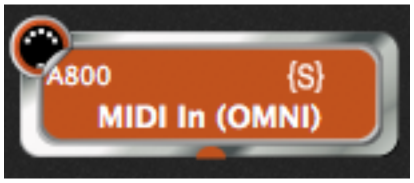
The keyword **initialization** defines the beginning of a callback that will be executed just once when a rackspace is initially loaded. In this example we simply assign a value to the variable and "print" its value. In the context of Gig Performer, printing means to show the item in a special window called the Script Logger. The keyword **end** completes the definition of this callback.

A more interesting example

```
// Declare a global variable
var    A800 : MidiInBlock  // A800 is a named block in Gig Performer

// Callback when a note event (on or off) is received
On NoteEvent (m : NoteMessage) From  A800
    SendNow(A800, m)  // Send the note to the A800 MidiIn block
    SendNow(A800, Transpose(m, 5)) // Also send a transposed note
End
```

Like before, the first thing we're doing is declaring a variable. But unlike the example above that just used a floating-point number, the variable A800 is declared with the type **`MidiInBlock`**. That type represents a Gig Performer MidiIn plugin block. Consequently, the name of the variable, in this case, A800, is significant. For this script to work there must be a MidiIn block in your rackspace and its scripting name, more properly known as a **handle**, must be A800. So in this example, we are binding a variable name to an entity in Gig Performer.



Note the handle name on the upper left and the {S} on the upper right, the latter indicating that the handle is available for use in GP Script

Next, we define a **callback**. A callback is a block of code that will run automatically when the event associated with that callback occurs. The callback above responds to MIDI note events (both NoteOn and NoteOff events) that arrive at the MidiIn block named A800. There are other callbacks that respond to other MIDI events such as Control Change events or Aftertouch. The actual message itself is accessed through the incoming parameter argument **m** which has the type **`NoteMessage`**.

Once you define a callback for a MIDI event you have full responsibility for that event. GP Script will not automatically send the MIDI event back to the MidiIn block which means that the MidiIn block will not forward the event to whatever synth plugins are connected to it. So if you want the incoming note to be played, you have to explicitly send it out, which we do with the statement **`SendNow(A800, m)`**. Immediately afterwards, we send the note again, after transposing it by 5 semitones. Voila, instant chords.

Basic concepts

GPScript is a programming language designed specifically for use with Gig Performer. Rather than integrating an existing language such as Lua, Python or ChaiScript and living with some compromises, we felt it worthwhile to have an essentially seamless environment that would immediately make sense. The clean syntax reflects that integration. GPScript is influenced more by Algol/Pascal.While we will assume for now that the reader has at least some basic understanding of programming there will be plenty of samples to help users to get up to speed quickly.

As mentioned above, GPScript is a strongly typed language; every variable must be declared along with a type. However, along with the usual types such as integers, strings, doubles and so forth, GPScript has built-in types that represent plugins, widgets, various kinds of LFOs and some special purpose types such as NoteTracker and ChordDictionary.

So as easily as you can write

```
var i : integer
```

to represent an integer value and perform operations such as addition and subtraction, you can write

```
var A800 : MidiInBlock
```

which represents a named MidiInBlock, presumably associated with an A800 MIDI keyboard controller. Now you can do operations such as manipulate incoming MIDI messages before sending them on to whatever audio plugin is connected to it.

GPScript includes the usual looping and conditional statements and of course you can define your own functions. There is also an extensive collection of efficient built-in functions you can leverage.

The latest list of built-in functions is best viewed online at
https://gigperformer.com/downloads/GPScript/SystemFunctionList.html

Scripts belong to rackspaces, meaning that you can have a completely separate script running in each of your rackspaces.

Tip: The top of the GP Script IDE has two buttons, that provide you with templates to help you get started as well as system function code completion.

Statements

Assignment

Syntax: <Identifier> ':=' <Expression>

Assign an expression to a variable. Example:

```
a := 42
b := a + 1

// You can also write an assignment using just the '=' character

Str = "I don't like C syntax"
```

For

Syntax: For <Assignment> ';' <BoolExpression> ';' <Assignment> Do <Statements> End

The for loop lets you iterate a statement block until the Boolean expression evaluates to false. It is often used to process an array of values.

Example

```
For i = 0; i < 10; i = i + 1 Do
   //statements here
End
```

While

Syntax: While <BoolExpression> Do <Statements> End

The While statement lets you loop though a block of statements until the Boolean expression evaluates to false. The important difference between a While loop and a For loop is that the former does not have an explicit clause to update the expression to be tested.

Example

```
Done = false
While !Done Do
   // Statements, one of which ultimately sets Done to true
End
```

Be very careful with both these looping constructs. If the condition does not ultimately evaluate to false, the loop will never end and Gig Performer will hang.

If

Syntax:

```
If <BoolExpression>
   Then <Statements>
  [ [Elsif <BoolExpression>
           Then <Statements>]
   Else <Statements> ]
End
```

This is the usual conditional test. The requirement for End prevents bugs due to the "dangling else" problem. The Elsif clause (as many as you need) is optional as is the Else clause.

Select

Syntax:

```
Select
   [<BoolExpression> do <Statements>]
```

```
End
```

The Select statement is a better option than the If statement when you need to test for many conditions. While similar in concept to the C "switch" statement or the Pascal "case" statement, the Select statement doesn't require each case to be a constant value.

Example:

```
select
   a > b do
      Print("A is greater than B")

   TimeNow() - previousTime > 1000 do
      Print("At least one second has passed")
      previousTime = TimeNow()

   true do
      Print("This executes if none of the other expressions were true")
end
```

Function calls

GP Script includes a large and growing collection of system functions for a variety of purposes such as math functions (rounding or scaling numbers), creation and modification of MIDI messages, management of function generators.

User functions

You can also write your own functions in GPScript

Syntax:

```
Function <name> ([<identifier> : <type> [, <identifier> : <type>]])
[Autotype] [Returns <type>]
<Local variable declarations>
<Statements>
End
```

Functions must have a unique name. Zero or more parameters can be defined and parameters are always passed in by value. Note that types such as Widgets, Plugins and dynamic arrays are objects and so what's really being passed by value is the reference (a pointer) to the object.

Functions can optionally return a value in which case the Returns clause must be present. If the Returns clause is present, then a variable called **`result`** exists in the function scope and whatever is assigned to that variable will be the return value.

There is no mechanism to return from the middle of a function.

Example

```
Function SumNumbers(a,b,c : integer) returns integer
   result = a + b + c
```

```
End

Function PlayMajorChord(keyboard : MidiInBlock, root : NoteMessage)
   SendNow(keyboard, root) // Original note
   SendNow(keyboard, Transpose(root, 4)) // The third
   SendNow(keyboard, Transpose(root, 7)) // The fifth
End
```

Operators

GP Script has the usual comparison operators (<, >, <=, >=, ==, !=) as well as boolean operators (And, Or, Not). Precedence rules are such that you very rarely require parentheses. GP Script also has a ternary operator following the style of Algol and, more recently, Haskell

```
Result = if widgetValue > 0.5 then 1 else 0
```

Note that the 'else' is not optional in the ternary operator

Callbacks - GPScript provides the following callbacks:

Initialization

```
 initialization
    // initialization code here. There can only be one of these
 end
```

Activation

```
on activate
   // Called whenever rackspace is activated
end

on deactivate
   // Called whenever we deactivate a rackspace
end
```

Variations

```
on variation ( oldVariation : integer, newVariation : integer)
   // Called when you switch to another variation
   // The parameters indicate which one you're in now
   // and from which one you came
end
```

MIDI events (Requires global variable associated with a Midi In Block)

```
On NoteOnEvent (m : NoteMessage) From  A800
   // Called whenever a Note On message is received
End

On NoteOffEvent (m : NoteMessage) From  A800
   // Called whenever a Note On message is received
End

On NoteEvent (m : NoteMessage) From  A800
   // Called whenever a Note On or Note Off message is received
   // Warning: this callback will not be invoked for NoteOn events
   // or for NoteOff events if you have defined
   // NoteOnEvent or NoteOffEvent respectively callbacks
End

On ControlChangeEvent (c : ControlChangeMessage) From A800
End

On PitchBendEvent (p : PitchBendMessage) From A800
End

On AftertouchEvent (a : AfterTouchMessage) From A800
   // This is channel pressure
End
```

```
On PolytouchEvent (p : PolyTouchMessage) From A800
   // This is note-specific pressure
End

On ProgramChangeEvent( p : ProgramChangeMessage) From A800
End
```

Constrained MIDI events

The Note events (Note, NoteOn, NoteOff) and the ControlChange event can optionally include a constraint that specifies either a range or an explicit sequence of numbers for which the callback is triggered. For example:

```
On NoteEvent (m : NoteMessage) Matching [C3..C4] From  A800
   // This event is only called if you play a note with
   // a MIDI note number that is in the range 60 to 72
End

On ControlChangeEvent (c : ControlChangeMessage) Matching 1,7,14 From  A800
   // This event is only called for controller numbers
   // 1, 7 or 14
End
```

When you used constrained events, you can have multiple events of the same kind as long as their ranges have no overlaps. The main purpose of this mechanism is to allow you to specify callbacks for specific purposes without having to handle all events in your script. For example, you might create a callback to handle key switches where the bottom octave of your keyboard is used for controlling some effects while notes in other octaves are handled normally.

If you create a callback with no constraints, then any subsequent callbacks with constraints will cause a compilation error because your non-constrained callback is handling all possible values.

On the other hand, if you create some callbacks with constraints and add a non-constrained callback afterwards, then that last callback will apply just to values that have not been specified in previously defined constrained callbacks.

Widget events (Requires global variable associated with widget)

```
On WidgetValueChanged (newValue : double) from SomeWidget
   // newValue represents a value between 0.0 and 1.0
End
```

Plugin events (Requires global variable associated with a plugin block)

```
On ParameterValueChanged(index : integer, value : double)
   // The parameter number and new value of the parameter
End
```

Generator events (Requires global variable associated with a generator)

```
On TimePassing(x : integer, y : double)
   // x is linear time based on the length (or frequency) of the generator
   // y is an amplitude with a value ranging between 0.0 and 1.0
   // The actual value for y at some position x depends on the kind
   // of generator and the settings being used
End
```

OSC callback

```
On OSCMessageReceived(message : OSCMessage) Matching StringConstant
   // StringConstant will be an explicit OSC Address to which this
   // callback will respond
   // Use the OSC functions in the system library to access the
   // arguments of the received message
End
```

Playhead events

```
On BeatChanged(barNumber : Integer, BeatNumber : Integer, tick : Integer)
   // Called every time the beatNumber increments
   // Note that barNumber and tick are currently unused
   // This is experimental
end
```

System events

```
On SystemEvent(newValue : double) Matching PlayheadStateChanged
   // Indicate that the playhead has been started (1.0) or stopped (0.0)
   // This is experimental
End
```

Reference

Comments

You can have both line comments and multiline comments. Single line comments start with a double slash and end at the line end.

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

Block comments start with /* and end with */ and they can be nested.

```
/*
   This is a
   multiline comment and
   /* you can nest multiline comments */
   as well
*/
```

Constants

GPScript supports integer, floating point, boolean and string constants (more on strings below). For convenience, MIDI note names can be used in place of integers. E.g, C3 is the same as 60, D#4 (or Eb4) is the same as 75.

Declarations

You can declare global variables and local variables. The general syntax for a declaration is

```
var identifier : type
```

If you want to declare multiple variables of the same type, just separate them with a comma

```
var i, j, k : integer
```

If you are declaring multiple variables one after the other, you do not need to repeat the var keyword (although you can)

```
var i : integer
    s : string
    n : NoteMessage
```

Global variable are defined outside of any callbacks or user-defined functions and are visible only from the point at which you declare them. Certain types such as Plugins, Widgets and function generators (ADSR, Ramp, etc.,) can only be declared at global scope.

The Type system

All variables must be declared with a type.

Primitive types

Primitive types include integer, double and boolean. With a couple of exceptions, operators for primitive types are part of the language syntax itself. For example, arithmetic operators (+,-,*,/,%) and comparators (<, <=, ==, !=, >, >=) can be used on both integers and doubles.

Opaque types

Opaque types are types whose behavior is controlled completely by system functions and you have no direct access to them. Variables that refer to “physical” widgets and plugin blocks instances of opaque types. Some opaque data types do not have any “physical” existence. These include function generators, the NoteTracker and the ChordRecognizer types. Over time, as new functionality is added to Gig Performer, more opaque data types are likely to be added.

Hybrid types

There are a couple of types that are treated specially, with some built-in operations even though they are otherwise opaque.

Strings

Language designers struggle with the best way to handle strings. Some leave all string processing up to their libraries (but still have language support to define string constants), others make strings be a special type with language support. GP Script adopts the latter approach. While there are as yet no string processing system functions (and it’s unclear what functions might actually be needed within the context of Gig Performer automation), the + operator is overloaded so that strings can be concatenated together. More importantly, if you start an expression with a string type, then the RHS of the + operator can be of type integer or double and the value will be automatically converted to a string type. This makes it easy to write such things as

```
Print(“The value is “ + i) // Display the value of i in the script log
window
```

and is mainly intended for debugging purposes although it can be convenient to use this mechanism to set the caption of a label or widget, e.g.,

```
SetWidgetCaption(w, “Value: “ + v)
```

Array

It is possible to have arrays of types (although currently you cannot have an array of arrays). Arrays currently have a maximum length of 128 (no prizes for guessing why that value). Array indexing is zero-based.

```
var mySmallArray : integer[32]
```

This creates an array that can contain 32 values. Attempts to reference the array outside the defined range will be trapped by the runtime system and the script terminated. While this adds a little cost, we feel that array range errors are sufficiently common that it's worth the cost. In the future, we may add an option to disable range checking.

You can pass arrays as dynamic parameters to functions and the system function **`size`** will return the current size of the array parameter.

```
Function IntSum(someArray : integer array) returns integer
   var s : integer // Track the sum
   index : integer

   for i = 0; i < size(someArray); i = i + 1 do
      s = s + someArray[i]
   end

   result = s // Return the result
End
```

Dynamic array

Rather than defining a constant size, you can add the keyword Array after any primitive type to create a dynamic array.

For example, the following declaration is allowed

```
Var
   Fader1, Fader2, Fader3, Fader4 : Widget
   Faders : Widget Array

Initialization
   Faders = [Fader1, Fader2, Fader3, Fader4] // Array initialization
End
```

Then you can use array indices to reference individual widgets. The **size** function works as expected and will return the current size of the array which of course is 4 in the above example.

Operators

Boolean operators

GPScript includes the usual Boolean operators but has some interesting additions to make scripting a little easier for users.

in

The **in** operator allows the left hand side, normally an integer or a floating point number to be tested against an integer or floating point respectively range. So you can write

```
if n in [C3..C4] then
```

The variable n would normally be an integer type. However, this will also work if n is a NoteMessage or a ControlChangeMessage and in these cases the note number or controller number will be used.