Lua API Extensions Reference

Overview

Lua には、そのコア機能を定義する標準ライブラリがありますが、インテリジェントモジュールと対話するには、追加の機能が必要です。 そのため、Symetrix は、インテリジェントモジュールとの対話に特に必要な機能を提供するカスタム Lua API（アプリケーションプログラミングインターフェイス）拡張機能のセットを作成しました。
次のページでは、Composer インテリジェントモジュールスクリプトで使用できる Lua のさまざまな Symetrix 拡張機能について説明します。

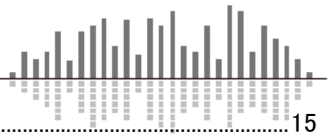
Conventions

コード例において、コード行の右側に次のように表示されます。:

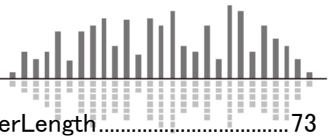
```
print("Ben")                    -->> Ben
ChangeLabel("Ben")              -- The label was changed to "Ben"
```

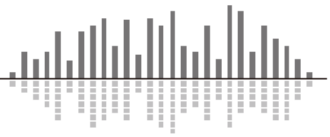
最初の行は、デバッグ出力ファイル(or デバッグ画面)に出力されます。
本書では print ステートメントの出力は "-->>" で示しますが、実際には.debug ファイルに出力されます。

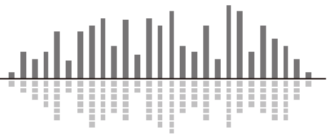
2 行目はコード内で何かを実行しますが、デバッグファイルへの出力はありません。
そのため、行の結果はコメントインジケータ "--" で記述されます。



INDEX







Controls

Overview

Controls API は、DSP モジュールの制御信号の入力ピンと出力ピンにアクセスするために使用されます。

ピン番号はインデックス 1 から始まります。

モジュール間の接続は、サンプルレートで更新されますが、どのモジュールも、より低いレートで信号にアクセスまたは変更できます。

スクリプトモジュールは、通常 4Hz より速く実行されないため、4Hz が制御ピンが読み取られる最大速度になります。

Controls I/O API は、通常オンラインでのみ役立つことに注意してください。 これは、モジュール間の接続がオンラインの場合にのみ機能するためです。 オフラインの場合、入力の読み取りと出力値の設定はできますが、接続されたモジュールとの間でデータがやり取りされることはありません。

Controls.Inputs

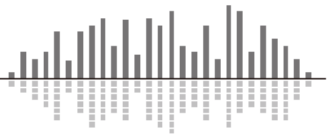
	Type	Range	Comment
Controls.Inputs[input].Value	Float	通常は 0-1 ですが、対応する入力ピンに接続されているものによって異なります	制御入力ピンに接続されている信号の値を読み取ります
Controls.Inputs[input].EventHandler	Callback		制御入力ピンに接続されている信号が変化したときにアクションを実行するために使用されます

・Controls.Inputs[input].Value

これは、対応する制御入力ピンに接続されている信号の値を読み取るために使用されます。

```
Pin1 = Controls.Inputs[1].Value                    --Pin1 = 0.78
```

すべての SymetrixControl モジュールは 0～1 の範囲の値を出力するため、入力ピンは通常、この範囲の浮動小数点値を受け取ります。 ただし、アーキテクチャは任意の 32 ビット浮動小数点数の値をサポートし、別のインテリジェントモジュールを作成して、異なる範囲の値を出力し、インテリジェントモジュール入力に接続することができます。 ただし、一般的には、インテリジェントモジュールが組み込みの Composer コントロールモジュールと正しくインターフェイスするように、可能であれば 0～1 の範囲を出力することをお勧めします。



・Controls.Inputs[input].EventHandler

値はタイマーループで呼び出すことができますが、定期的に変更されない入力の場合、これは非効率的です。

EventHandler を使用すると、定期的にポーリングして変更をチェックする代わりに、制御入力ピンからの変更が発生した場合にのみ処理できます。

EventHandler には、対応する制御入力が変更されるたびに呼び出される関数が割り当てられます。

これは、Symetrix ハードウェアで実行している場合、最大 4Hz で更新されます。

これは 2 つの方法で行うことができます。

```
function PinChanged()
    pin[1] = Controls.Inputs[1].Value
end

Controls.Inputs[1].EventHandler = PinChanged          --pin[1] = 0.78
```

または、匿名関数を使用することもできます:

```
Controls.Inputs[1].EventHandler = function()
Pin[1] = Controls.Inputs[1].Value
end
```

どちらの例でも、Pin1 が変更されると、内部 Pin テーブルが更新されます。 他の入力制御ピンにはイベントハンドラー関数があります。 それらは必要に応じて類似または完全に異なる場合があります。

Controls.Outputs

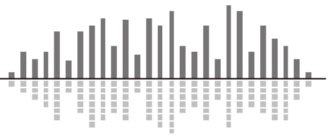
	Type	Range	Comment
Controls.Outputs[output].Value	Float	通常は 0-1 ですが、出力ピンに接続されているモジュールに必要なものによって異なります	制御出力ピンに値を書き込みます

・Controls.Outputs[output].Value

これは、対応する制御出力ピンにフロート値を書き込むために使用されます。

```
Controls.Outputs[1].Value = 10.5                  --Output Pin1 = 10.5
```

上記のように、値は、出力ピンに接続するモジュールの必要に応じて任意のフロートにすることができます。 すべての Symetrix 制御モジュールは 0～1 の範囲の入力値を想定しているため、インテリジェントモジュールの出力ピンは通常この範囲の浮動小数点値を送信します。 ただし、別のインテリジェントモジュールを作成して、異なる範囲の値を受信し、インテリジェントモジュールの出力に接続することもできます。 ただし、一般的には、インテリジェントモジュールが組み込みの Composer コントロールモジュールと正しくインターフェイスするように、可能であれば 0～1 の範囲を出力することをお勧めします。



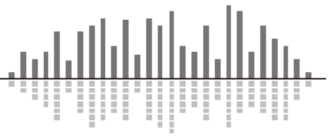
Usage Examples

次の例は、これらの使用方法を示しています:

・Example 1 – 二つの入力を比較する

次の例は、2 つの入力ピンの値が変化したときに比較する方法と、比較結果に基づいて設定された出力ピンの値を示しています。

```
function PinChanged()
    state1 = Controls.Inputs[1].Value
    state2 = Controls.Inputs[2].Value
    if (state1 > .5) and (state2 > .5) then
        output = 1.0
    else
        output = 0.0
    end
    Controls.Outputs[1].Value = output
end

Controls.Inputs[1].EventHandler = PinChanged
Controls.Inputs[2].EventHandler = PinChanged
```



・Example 2 – ポーリング

イベントハンドラーが最も頻繁に使用されますが、ピンが定期的に変更されることが予想される場合や、スクリプトに UI で Named.Controls をポーリングするためのタイマーが既に含まれている場合は、コントロールピンもポーリングすることをお勧めします。

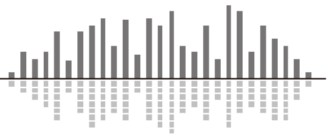
以下の例では、例 1 でポーリングを使用するように変換しています。

```
function TimerClick ()
    state1 = Controls.Inputs[1].Value
    state2 = Controls.Inputs[2].Value

    if (state1 > .5) and (state2 > .5) then
        output = 1.0
    else
        output = 0.0
    end

    Controls.Outputs[1].Value = output
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(.25)
```

・Example 3 – 複数の I/O を繰り返し処理する

同様の処理を必要とし、ポーリングを使用している入力や出力が多数ある場合は、次のコードを使用してそれらを反復処理できます。

```
function TimerClick ()
    for pin, state in ipairs(Controls.Inputs) do
--iterate through each input pin
        print("Input pin " .. pin .. " value: " .. state.Value)
--print the pin and value
--Do processing for outputs using “pin” variable as index
        Controls.Outputs[pin].Value = state.Value/2
    end
end
MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(1)
```



Device

Overview

Device API は、オンラインの場合はメインのデバイスファームウェア、オフラインの場合は Composer によって維持されるさまざまな定数（および疑似定数）へのスクリプトアクセスを可能にするグローバル変数テーブルです。

これは、インテリジェントモジュールを特定のサードパーティの Dante デバイスに関連付けることができる TCP / UDP 通信に特に役立ちます。

Device.

	Type	Comment
Device.Offline	Bool	Composer でオフラインで実行している場合は True

・Device.Offline

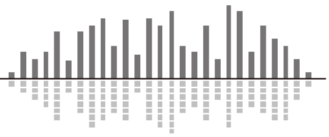
Composer がオフラインのときにスクリプトが実行されている場合は、「true」を返します。

Composer がオンラインのときにスクリプトが実行されている場合、「false」を返します。

```
print(Device.Offline)                -->> false
```

これは、オンラインまたはオフラインの場合にさまざまなことを行うために非常に一般的に使用されます。

```
if Device.Offline then
    --Do some stuff when Offline
else
    --Do different stuff when Online
end
```



Device.LocalUnit

Device.LocalUnit API は、スクリプトが実行されているデバイスに関する情報を返します。

	Type	Comment
Device.LocalUnit.ControlIP	String	制御に使用されるネットワークアダプタの IP アドレス
Device.LocalUnit.AudioIP	String	ネットワークオーディオに使用されるネットワークアダプタの IP アドレス
Device.LocalUnit.Name	String	スクリプトを実行しているデバイスのネットワーク名
Device.LocalUnit.Type	Integer	スクリプトを実行しているデバイスのハードウェアタイプ
Device.LocalUnit.TypeName	String	スクリプトを実行しているデバイスのハードウェアタイプ名

・Device.LocalUnit.ControlIP

制御に使用されるネットワークアダプタの IP アドレスを返します。 オンラインの場合、これはスクリプトが実行されているデバイスの実際の制御 IP アドレスになります。

```
print(Device.LocalUnit.ControlIP)          -->> 192.168.0.10
```

IP アドレスが解決される前にオンラインの場合、これはゼロになります。 スクリプトは、ネットワーク接続がない場合でも、最終的にリンクローカルアドレスを取得するため、デバイスがまだ起動を完了していないことを示す指標として、ゼロを探すことができます（そして探す必要があります）。

Composer がオフラインの場合、Intelligent Module Options ウィンドウで制御用に指定されたネットワークアダプタの IP アドレスを返します。

・Device.LocalUnit.AudioIP

ネットワークオーディオに使用されるネットワークアダプタの IP アドレスを返します。 オンラインの場合、これはスクリプトが実行されているデバイスの実際のオーディオネットワーク IP アドレスになります。

```
print(Device.LocalUnit.AudioIP)            -->> 169.254.1.21
```

IP アドレスが解決される前にオンラインの場合、これはゼロになります。 スクリプトは、ネットワーク接続がない場合でも、最終的にリンクローカルアドレスを取得するため、デバイスがまだ起動を完了していないことを示す指標として、ゼロを探すことができます（そして探す必要があります）。

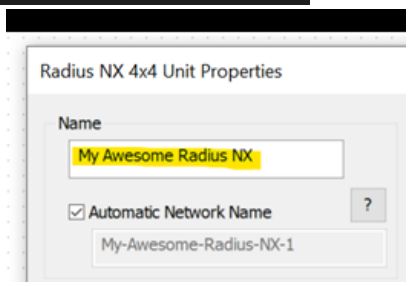
Composer がオフラインの場合、Intelligent Module Options ウィンドウでネットワークオーディオ用に指定されたネットワークアダプタの IP アドレスを返します。

・Device.LocalUnit.Name

スクリプトが実行されているデバイスの名前を返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。

```
print(Device.LocalUnit.Name)                --->> My Awesome Radius NX-1
```

この名前は、サイト内のすべてのデバイスで一意になります。 これは、ユーザーのデバイス名（Unit Properties ダイアログで編集可能）を取得し、Composer が提供する数値を最後に追加することによって行われます。 上記の例の場合、「-1」が追加されます。



・Device.LocalUnit.Type

スクリプトが実行されているデバイスのタイプのコードを返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。

```
print(Device.LocalUnit.Name)                --->> 113
```

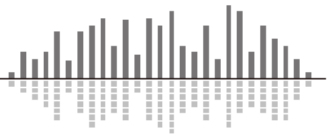
以下は、インテリジェントモジュールをサポートするすべての Symetrix デバイスのコードリストです。:

- Invalid = 0
- Edge = 36
- Solus NX 4x4 = 40
- Solus NX 8x8 = 41
- Solus NX 16x16 = 42
- Radius 12x8 = 43
- Radius 12x8 AEC = 46
- Radius 12x8 EX = 64
- Prism 4x4 = 71
- Prism 8x8 = 72
- Prism 12x12 = 73
- Prism 16x16 = 74
- Prism 0x0 = 104
- Radius NX 12x8 = 110
- Radius NX 4x4 = 113

・Device.LocalUnit.TypeName

スクリプトが実行されているデバイスの名前を返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。

```
print(Device.LocalUnit.TypeName)                --->> Radius NX 4x4
```



Device.RemoteUnit

Device.RemoteUnit API は、関連するユーザーライブラリ Dante デバイスに関する情報を返すため、これらのタイプのインテリジェントモジュールでのみ使用する必要があります。 スタンドアロンインテリジェントモジュールのスクリプトで使用された場合、Device.RemoteUnit は nil を返すため、API 呼び出しのいずれかでエラーが発生します。 これら 2 種類のインテリジェントモジュールの詳細については、カスタムインテリジェントモジュールの作成を参照してください。.

	Type	Comment
Device.RemoteUnit.IP	String	既知の場合、Dante ユニットの制御インターフェースの IP アドレス
Device.RemoteUnit.Name	String	制御されている Dante ユニットの名前
Device.RemoteUnit.Type	Integer	制御されている Dante ユニットのハードウェアタイプのコード
Device.RemoteUnit.TypeName	String	制御対象の Dante ユニットのハードウェアタイプ名
Device.RemoteUnit.DanteIP	String	Dante ユニットのオーディオインターフェースの IP アドレス
Device.RemoteUnit.DanteMAC	String	XX:XX:XX:XX:XX の形式の Dante ユニットの MAC アドレス
Device.RemoteUnit.DanteName	String	制御されている Dante ユニットの Dante 名
Device.RemoteUnit.DanteModel	String	制御対象の Dante ユニットの型名
Device.RemoteUnit.DanteMfg	String	制御対象の Dante ユニットのメーカー名

・Device.RemoteUnit.IP

既知の場合、関連する Dante デバイスの制御に使用されるネットワークアダプターのドット付き IP アドレスを返します。 オンラインの場合、これは関連する Dante デバイスの実際の制御 IP アドレスになります。

```
print(Device.RemoteUnit.IP)          -->> 192.168.0.10
```

これはほとんどのリモートデバイスでは不明であり、制御 IP は他の方法で決定し、スクリプトで使用するためにユーザーが手動で入力する必要があることに注意してください。 この場合、API は「0.0.0.0」を返します。

・Device.RemoteUnit.Name

関連する Dante デバイスのネットワーク名を返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。 これは、サイト内のすべてのデバイスに固有です。

```
print(Device.RemoteUnit.Name)              -->> My Cool Dante Mic
```

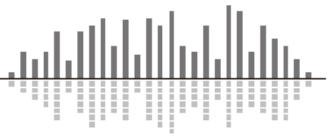
これは、Composer で割り当てられたネットワーク名に対応しており、「Unit Properties」ダイアログから自動または手動で設定できます。

・Device.RemoteUnit.Type

関連する Dante デバイスのタイプのコードを返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。

```
print(Device.RemoteUnit.Type)              -->> 94
```

すべてのユーザーライブラリ Dante デバイスに単一のコード「94」が使用されます。したがって、すべてのデバイスがこのコードを返します。



・Device.RemoteUnit.TypeName

関連する Dante デバイスのタイプの名前を返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。 これは、同じタイプのすべてのデバイスで同じになります。

```
print(Device.RemoteUnit.TypeName)          -->> Audio-Technica-ATND971
```

・Device.RemoteUnit.DanteIP

関連する Dante デバイスの Dante ネットワークに使用されるネットワークアダプタの IP アドレスを返します。
これは、オフラインまたはオンラインで実行している場合も同じように動作します。

```
print(Device.RemoteUnit.DanteIP)           -->> 169.254.1.21
```

・Device.RemoteUnit.DanteMAC

関連する Dante デバイスの Dante ネットワークに使用されるネットワークアダプタの MAC アドレスを返します。
これは、オフラインまたはオンラインで実行している場合も同じように動作します。

```
print(Device.RemoteUnit.DanteMAC)          -->> 12:34:56:78:90
```

・Device.RemoteUnit.DanteName

関連する Dante デバイスの Dante 名を返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。 これは、サイト内のすべてのデバイスに固有です。

```
print(Device.RemoteUnit.DanteName)         -->> My Cool Dante Mic
```

これは、DanteController で割り当てられた名前です。

・Device.RemoteUnit.DanteModel

関連する Dante デバイスの Dante モデル名を返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。 これは、同じモデルのすべての製品で同じになります。

```
print(Device.RemoteUnit.DanteModel)        -->> ATND971
```

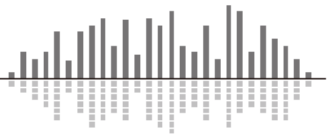
これは、DanteController の Status タブにあるモデル名です。

・Device.RemoteUnit.DanteMfg

関連する Dante デバイスの Dante 製造元名を返します。 これは、オフラインまたはオンラインで実行している場合も同じように動作します。 これは、同じメーカーのすべての製品で同じになります。

```
print(Device.RemoteUnit.DanteMfg)          -->> Audio-Technica
```

これは、DanteController の Status タブにあるメーカー名です。



Usage Examples

次の例は、これらの使用方法を示しています

:・**Example 1 – オンライン/オフラインを確認し UDP を開く**

Device API が使用される最も一般的なことの 1 つは、スクリプトがオフラインで実行されているかオンラインで実行されているかを判断し、UDP ソケットを開くために適切な IP アドレスを使用することです。

```
if Device.Offline then
    MyUdp:Open("192.168.1.101", 0)                --Use computer IP
else
    MyUdp:Open(Device.LocalUnit.ControlIP, 0)     --Use device IP
end
```

・Example 2 – IP アドレスが使用可能かどうかチェックする

IP アドレスはまだ有効ではない可能性があるため、使用する前に確認する必要があります。

```
--state
validIP = false

function TimerClick ()
       --Check if IP was previously marked as valid
    if validIP ~= true then
                --If not yet valid, check if IP is Valid
        if Device.LocalUnit.ControlIP ~= 0.0.0.0 then
            --If valid, mark it as so for future use
        validIP = true
        end
    --if not valid, try again next time timer fires
    else
    --do communications using the validated IP address
    end
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(.5)
```

HttpClient

Overview

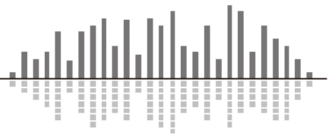
HttpClient は、カスタム HTTP プロトコルを介して他のデバイスや Web サービスとの通信を可能にするメソッドのグローバルテーブルです。HttpClient API は、制御ネットワーク上でのみ機能します。

HttpClient API は、Radius NX、Edge、Prism デバイスと互換性がありますが、Solus NX DSP では実行できません。

HttpClient.

HttpClient オブジェクトには、以下のメソッドとプロパティがあります。

Parameters	Return Type	Description
HttpClient.Download	Bool	特定の URL からテキストデータを受信する際に使用します。HTTP または HTTPS で特定の URL からテキストデータを受信するために使用します。 テキストデータは、テキストで表現された任意のコンテンツです。 コマンドを受け付けた場合は、true を返します。
HttpClient.Upload	Bool	HTTP または HTTPS を使用して、特定の URL にテキストデータを送信するために使用します。テキストデータは、文字列化された JSON オブジェクトまたはプレーンテキストです。 コマンドを受け付けた場合は、true を返します。
HttpClient.CreateURL	String	ホスト名とポート番号、パス、クエリを組み合わせて URL を作成するオプションのテーブルを組み合わせて、ダウンロードやアップロードのメソッドで使用する 1 つのフォーマットされた URL を作成します。
HttpClient.EncodeParams	String	パラメータと値のテーブルを 1 つの文字列にまとめて 1 つの文字列にまとめて、URL を作成します。
HttpClient.EncodeString	String	スペースなど、URL に使用できない文字を含む文字列を、URL セーフに変換します。URL に使用できないスペースやその他の文字を含む文字列を、URL セーフな エンコーディングに変換します。
HttpClient.DecodeString	String	スペースなど、URL に含まれないその他の文字を取り除いてエンコードされた文字列を、読みやすい形に変換します。



HttpClient.Download

Download メソッドは、Web サイト上のリモートテキストコンテンツを取得するためのものです。1 つの文字列として表現できるコンテンツにのみアクセスできます。
文字列化された JSON オブジェクトをダウンロードし、JSON API のデコード・メソッドを使用してテーブルに変換することができます。

Download メソッドは、コールが受け入れられたかどうかを示す Bool 値を返します。
コールバックなしで 8 回以上のダウンロードまたはアップロードコールが開かれている場合は false が返され、そうでない場合は true が返され、最終的にコールバックが呼び出されます。
レスポンスやエラーを受け取るには、EventHandler コールバック関数を指定する必要があります。

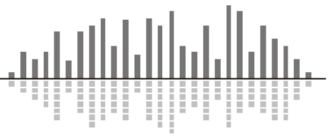
最大 8 つのダウンロードおよびアップロードコールが、すべてのモジュールを合わせて、レスポンスまたはタイムアウトを待っている可能性があります。
これらはすべて 1 つのモジュールから来ることもあれば、複数のモジュールから来ることもあります。

レスポンスやタイムアウトが発生すると、そのスロットは別のアクセスのために空きます。
この制限を超えた場合、ダウンロードやアップロードのコールは直ちに失敗し、そのコマンドのコールバックは発生しません。

Parameters	Type	Required	Description
Url	string	Required	ダウンロード先の URL
Headers	table	Optional	ヘッダのテーブルで、それぞれが属性の割り当て
User	string	Optional	認証されたサイトのユーザー名
Password	string	Optional	認証されたサイトのパスワード
Timeout	number	Optional	ダウンロード操作のタイムアウトを秒単位で指定
EventHandler	function	Required	完了時やエラー時に呼び出されるコールバック関数

- **Url**

  適切な Url パラメータは、"http://www.symetrix.co"のようなフォーマットされたウェブドメイン名で構成されます。
  アクセスする URL を作成するには、CreateUrl の使用を検討してください。
  URL の最大長は 1023 文字で、それ以上の長さの文字は警告なしに切り捨てられます。
  なお、CreateUrl メソッドの出力は 4095 文字です。
  Download メソッドは Url パラメータを必要とします。
  しかし、Url パラメータが空白であっても、コマンドは拒否されません。
  代わりに存在しない URL や不正な URL に起因するエラーは、コールバックにエラーとして返されます。



- **Headers**

  Headers パラメータは、それ自体が、必要なすべてのヘッダのテーブルです。
  各ヘッダには、["content-type"] = "text/*"のような適切なフォーマットの配列要素があります。
  ヘッダは必須ではありません。
  各ヘッダは、キーと値で構成され、キーと値は最大 255 文字までとなり、それ以上は警告なしに切り捨てられます。一般的なヘッダの詳細については、MDN をご覧ください。

- **User**

  User パラメータは、必要に応じて認証用のユーザー名を指定します。
  デフォルトでは、User の値はありません。User を指定しない場合、入力された Password は使用されません。
  ユーザーの文字数は 31 文字までで、それ以上の文字数は警告なしに切り捨てられます。

- **Password**

  Password パラメータは、必要に応じて認証用のパスワードを設定します。
  デフォルトでは、User の値はありません。User を指定しない場合、入力されたパスワードは使用されません。
  パスワードの文字数は 31 文字までで、それ以上の文字数は警告なしに切り捨てられます。

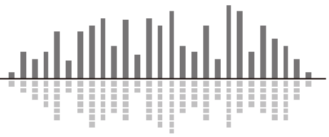
- **Timeout**

  Timeout パラメータは、タイムアウトエラーを返すまでにレスポンスを待つ時間(秒数)を指定します。
  任意の数値を指定できますが、浮動小数点は次の整数秒に切り上げられます。
  指定しない場合、デフォルトのタイムアウトは 30 秒です。

- **EventHandler**

  EventHandler パラメータは、レスポンスを受信したとき、エラーが発生したとき、またはタイムアウトが発生したときに呼び出されるべき関数です。
  EventHandler のコールバック関数は、以下の形式でなければなりません。
  EventHandler (table Parameters, number ReturnCode, string Data, string Error, table Headers)

Parameters	Type	Description
Parameters	table	ダウンロードメソッドに送信されるパラメータテーブル
ReturnCode	number	送信先 URL から受信したステータスコード
Data	string	送信先 URL から返されるデータを含む文字列
Error	string	発生したエラーの状態を表す文字列
Headers	table	送信先 URL から返されるヘッダのテーブル



- **Parameters**

  引数 Parameters には、HttpClient.Download メソッドに送られた Parameters テーブルが格納されます。このテーブルは、EventHandler から戻るまで保存されます。

- **Return Code**

  ReturnCode は、送信先の URL から受け取った番号です。
  通常、リクエストが成功したレスポンスを示す 200 が返されますが、返される可能性のある HTTP ステータスコードは数多くあります。通常、EventHandler はコードに応じて異なるアクションを実行します。
  https://ja.wikipedia.org/wiki/HTTP ステータスコード

- **Data**

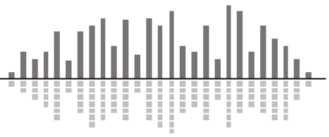
  引数 Data には、返されるデータを含む文字列を指定します。文字列が文字列化された JSON オブジェクトの場合は、json ライブラリを使って変換して返すことができます。
  返される文字列の最大長は 64000 文字です。
  内容がこれより大きい場合、Data は nil となり、エラーテキストが返されますが、ReturnCode はサイトから返されたリターンコードがそのまま返されます。
  注意：大きな文字列は Lua での処理効率が悪いため、インテリジェントモジュールでの処理時間が長すぎて実行エラーになる可能性があるので、注意して使用する必要があります。

- **Error**

  引数 Error には、発生したエラー状態を表す文字列を指定し、エラーがない場合は nil を指定します。
  エラーは、コンピュータ（スクリプトをオフラインで実行する場合）または Symetrix デバイスのファームウェア（スクリプトをオンラインで実行する場合）によって生成されます。
  エラー文字列の最大長は 255 文字です。表示される一般的なエラーには次のようなものがあります。
  - "Download returned too much data to handle!"
    （ダウンロードしたデータが多すぎて処理できませんでした）
  - "Download returned too much headers data to handle!"
    （ダウンロードしたヘッダーデータが多すぎて処理できませんでした）
  - "Upload returned too much data to handle!"
    （アップロードのデータ量が多すぎて処理できませんでした）
  - "Download returned too much headers data to handle!"
    （ダウンロードで返されたヘッダーデータが多すぎて処理できませんでした）
  - "Timeout waiting for response from Http request service!"
    （Http リクエストからのレスポンスがタイムアウトしました）

- **Headers**

  Headers 引数は、成功した場合に宛先 URL から返されるヘッダのテーブルです。
  一般的なヘッダの詳細については、MDN をご覧ください

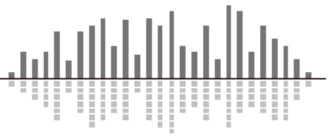


HttpClient.Download の例

```
-- callback function
function Response(Table, ReturnCode, Data, Error, Headers)
    print(string.format("URL requested = '%s'.", Table.Url))
    if (200 == ReturnCode) then
        print("Success!")
        print(string.format("Data returned = '%s'", Data))
    else
        print(string.format("Failed, error code #%d, '%s'",
ReturnCode, Error))
    end
end

HttpClient.Download { Url = "https://www.symetrix.co", Headers =
{ ["Content-type"] = "text/*" },
Timeout = 3, EventHandler = Response }
```

このコードでは、以下の返答が得られます。

```
URL requested = 'https://www.symetrix.co'.
Success!
Data returned = 'nil'
```



HttpClient.Upload

Upload メソッドは、ウェブサイト上でホストされているリモートコンテンツをテキストコンテンツで修正することができます。送信できるのは、単一の文字列で表現できるコンテンツのみです。
JSON オブジェクトは、JSON API の encode メソッドで文字列化してからアップロードすることができます。

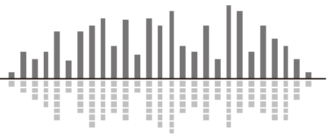
Upload メソッドは、コールが受け入れられたかどうかを示す Bool 値を返します。
コールバックなしで 8 つ以上の Download または Upload コールが開かれている場合、false が返されます。
false が返されますが、そうでない場合は true が返され、最終的にコールバックが呼び出されます。
応答やエラーを受け取るには、EventHandler コールバック関数を指定する必要があります。

最大 8 つのダウンロードおよびアップロードコールが、すべてのモジュールを合わせて、レスポンスまたはタイムアウトを待っている可能性があります。
これらはすべて 1 つのモジュールから来ることもあれば、複数のモジュールから来ることもあります。

レスポンスやタイムアウトが発生すると、そのスロットは別のアクセスのために空きます。
この制限を超えた場合、ダウンロードやアップロードのコールは直ちに失敗し、そのコマンドのコールバックは発生しません。

Upload メソッドは Parameters 引数を必要とし、それ自体が引数のテーブルです。
テーブルが提供されていない場合、この関数は nil を返します。
Parameters テーブルには、URL、Headers、User、Password、Data、Method、Timeout、EventHandler が含まれます。

Parameters	Type	Required	Description
Url	string	Required	アップロード先の URL
Headers	table	Optional	ヘッダのテーブルで、それぞれが属性の割り当て
User	string	Optional	認証されたサイトのユーザー名
Password	string	Optional	認証されたサイトのパスワード
Date	string	Required	アップロードするデータ
Method	string	Optional	アップロードメソッド
Timeout	number	Optional	ダウンロード操作のタイムアウトを秒単位で指定
EventHandler	function	Required	完了時やエラー時に呼び出されるコールバック関数



- **Url**

  適切な Url パラメータは、"http://www.symetrix.co"のようなフォーマットされたウェブドメイン名で構成されます。
  アクセスする URL を作成するには、CreateUrl の使用を検討してください。
  URL の最大長は 1023 文字で、それ以上の長さの文字は警告なしに切り捨てられます。
  なお、CreateUrl メソッドの出力は 4095 文字です。
  Download メソッドは Url パラメータを必要とします。
  しかし、Url パラメータが空白であっても、コマンドは拒否されません。
  代わりに存在しない URL や不正な URL に起因するエラーは、コールバックにエラーとして返されます。

- **Headers**

  Headers パラメータは、それ自体が、必要なすべてのヘッダのテーブルです。
  各ヘッダには、["content-type"] = "text/*"のような適切なフォーマットの配列要素があります。
  ヘッダは必須ではありません。
  各ヘッダは、キーと値で構成され、キーと値は最大 255 文字までとなり、それ以上は警告なしに切り捨てられます。一般的なヘッダの詳細については、MDN をご覧ください。

- **User**

  User パラメータは、必要に応じて認証用のユーザー名を指定します。
  デフォルトでは、User の値はありません。User を指定しない場合、入力された Password は使用されません。
  ユーザーの文字数は 31 文字までで、それ以上の文字数は警告なしに切り捨てられます。

- **Password**

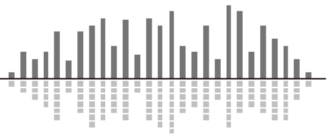
  Password パラメータは、必要に応じて認証用のパスワードを設定します。
  デフォルトでは、User の値はありません。User を指定しない場合、入力されたパスワードは使用されません。
  パスワードの文字数は 31 文字までで、それ以上の文字数は警告なしに切り捨てられます。

- **Data**

  Data 引数には、ウェブサイトに送信するテキストデータを文字列の形で指定します。
  Upload メソッドでは、Data 引数が必要ですが、空の文字列や nil でも OK です。送信される文字列の最大長は 64000 文字となります。これよりも大きい場合は、送信前に警告なしに切り捨てられます。
  注意：大きな文字列は Lua での処理効率が悪く、インテリジェントモジュールで処理時間がかかりすぎて実行エラーになる可能性があるため、注意して使用する必要があります。

- **Method**

  Method パラメータは、アップロードに使用されるメソッドを指定します。通常は「POST」ですが、サーバーがサポートしている場合は「PUT」でもかまいません。nil または空文字列を指定すると、POST が使用されます。それ以外のメソッドは、HTTP エンジンに直接渡されます。メソッドは 31 文字までで、それ以上の文字数は警告なしに切り捨てられます。



- **Timeout**

  タイムアウトエラーを返すまでの応答待ちの秒数です。任意の数値を指定できますが、浮動小数点は切り上げられ、次の整数秒になります。指定しない場合、デフォルトのタイムアウトは 30 秒です。

- **EventHandler**

  EventHandler パラメータは、レスポンスを受信したとき、エラーが発生したとき、またはタイムアウトが発生したときに呼び出されるべき関数です。

  EventHandler のコールバック関数は、以下の形式でなければなりません。

  EventHandler (table Parameters, number ReturnCode, string Data, string Error, table Headers)

Parameters	Type	Description
Parameters	table	ダウンロードメソッドに送信されるパラメータテーブル
ReturnCode	number	送信先 URL から受信したステータスコード
Data	string	送信先 URL から返されるデータを含む文字列
Error	string	発生したエラーの状態を表す文字列
Headers	table	送信先 URL から返されるヘッダのテーブル

- **Parameters**

  引数 Parameters には、HttpClient.Download メソッドに送られた Parameters テーブルが格納されます。このテーブルは、EventHandler から戻るまで保存されます。

- **Return Code**

  ReturnCode は、送信先の URL から受け取った番号です。

  通常、リクエストが成功したレスポンスを示す 200 が返されますが、返される可能性のある HTTP ステータスコードは数多くあります。通常、EventHandler はコードに応じて異なるアクションを実行します。

  https://ja.wikipedia.org/wiki/HTTP ステータスコード

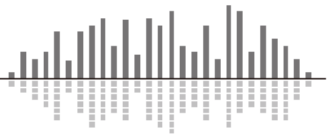
- **Data**

  引数 Data には、返されるデータを含む文字列を指定します。文字列が文字列化された JSON オブジェクトの場合は、json ライブラリを使って変換して返すことができます。

  返される文字列の最大長は 64000 文字です。

  内容がこれより大きい場合、Data は nil となり、エラーテキストが返されますが、ReturnCode はサイトから返されたリターンコードがそのまま返されます。

  注意：大きな文字列は Lua での処理効率が悪いため、インテリジェントモジュールでの処理時間が長すぎて実行エラーになる可能性があるので、注意して使用する必要があります。

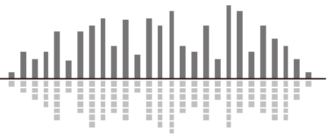


- **Error**

  Error 引数には、発生したエラー状態を表す文字列、またはエラーなしの場合は nil を指定する。エラー文字列の最大長は 255 文字である。

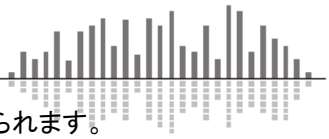
- **Headers**

  Headers 引数は、成功した場合に宛先 URL から返されるヘッダのテーブルです。一般的なヘッダの詳細については、MDN をご覧ください



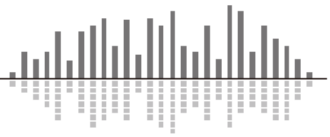
HttpClient.Upload の例

```
--Use Post Test Server to test upload
--https://ptsv2.com/s/howitworks.html
ptsUrl = "Insert your Post Test URL Here"
-- Upload callback function
function UploadResponse(Table, ReturnCode, Data, Error, Headers)
      print("")
      print("Upload Callback:")
      print(string.format("URL requested = '%s'.", Table.Url))
      print(string.format("Data uploaded = '%s'.", Table.Data))
      print("Return Code: "..ReturnCode)
      print("Headers:")
      if (Headers ~= nil) then
            for headerName, headerValue in pairs(Headers) do
                  print(string.format( "¥t"..headerName..": "..headerValue ) )
            end
      else
            print("¥tNone")
      end
      if (200 == ReturnCode) then
            print("Upload Success!")
            print(string.format("Data returned = '%s'", Data))
      else
            print(string.format("Failed, error code#%d, '%s'", ReturnCode, Error))
      end
end
--Do Upload
HttpClient.Upload { Url = ptsUrl, Method = "POST", Data = "test message", Timeout =
3, EventHandler = UploadResponse }
```



このコードは、有効な Post Test Server の URL を指定して実行すると、以下の返答を得られます。

```
Upload Callback:
URL requested = 'http://ptsv2.com/t/rx49p-1611678942'.
Data uploaded = 'test message'.
Return Code: 200
Headers:
      X-Cloud-Trace-Context: 0365d7484760d3389ff39ce47de2d4ff
      Content-Type: text/html; charset=utf-8
      Content-Length: 5229
      Date: Tue, 26 Jan 2021 16:36:12 GMT
      Vary: Accept-Encoding
      Server: Google Frontend
Upload Success!
Data returned = '<!DOCTYPE html>
…
```



HttpClient.CreateURL

ホスト名とポート番号、パス、クエリの組み合わせから URL を作成するオプションのテーブルを組み合わせ、ダウンロードとアップロードのメソッドで使用するための単一のフォーマットされた URL にします。スクリプトは文字列を自分で作成することができ、役に立たない場合はこのメソッドを使用する必要はありません。

CreateUrl メソッドは、構築された Lua 文字列という正確に 1 つの値を返します。エラーが発生した場合は、nil が返されます。返される文字列の最大長は 4095 文字です。

CreateUrl メソッドには Parameters 引数が必要で、それ自体が引数の表になっています。テーブルが提供されない場合、関数は nil を返します。Parameters テーブルには、Host、Port、Path、および Query オプションを含めることができ、それぞれパラメータ名でインデックス付けされます

Parameters Table Index	Type	Required	Description
Host	string	Required	ホスト名（例：http://symetrix.co）
Port	number	Optional	使用するポート番号
Path	string	Optional	アクセスするホストネーム内のパス
Query	table	Optional	URL にエンコードするパラメータと値のテーブル
Encode	bool	Optional	クエリをエンコードするか、URL に直接テキストを使用するかを選択

- **Host**

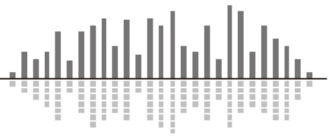
  Host オプションには、通常、"http://"または "https://"のプレフィックスを含む必要があります。しかし、接頭辞があるかどうかは検証されません。必要であれば、メソッドが呼ばれた後にプレフィックスを追加することができます。DNS 解決は、Host オプションでは行われない。Host オプションには、スペースやその他の特殊文字を含めないようにします。これは、URL の自動エンコードが行われないためです。Host の最大長は 1023 文字です。これより長い文字列は切り捨てられます。

- **Port**

  Port オプションは、URL の中に任意のポート番号を含めることができます。ポート番号が整数でない場合、または 1～65535 の範囲外である場合は無視されます。

- **Path**

  Path オプションは、ホスト名空間内の URL への絞り込みパスを指定することができます。Path オプションは、先頭のスラッシュを持つ必要はない。空でないパスの先頭にスラッシュがなく、空でない URL の末尾にスラッシュがない場合、URL 構築時に必要に応じて追加されます。スペースやその他の特殊文字は URL エンコードされ、スペースが"%20 "に置き換えられるように、これらの文字を有効な URL 文字と同等に置き換えます。例外として、URL エンコードされない「:」文字があります。これらの文字が必要な場合は、手動でエンコードする必要があります。パスの最大長は 255 文字です。



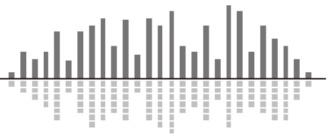
- **Query**

  Query オプションは、1 つまたは複数のインデックス付きパラメータを含むオプションのテーブルです。各パラメータは、インデックスと値です。必要な"? "憲章は、メソッドによって最初のクエリに前置され、"&"文字は、必要に応じて項目間にメソッドによって追加されます。空のクエリテーブルは、テーブルがないのと同じです。各キーの最大長は 255 文字です。長いキーは切り捨てられます。各値の最大長は、255 文字です。長い値は切り捨てられます。キーと値のペアを追加すると、URL の最大長を超える場合は、追加されません。キーと値はデフォルトで URL エンコードされていますが、後述のエンコードオプションで上書きすることができます。

- **Encode**

  Encode オプションは bool 値で、Query index および Query value フィールドのスペースやその他の特殊文字を URL エンコードするかどうかを制御します。指定されていない場合は、true が使用されます。
  これは、クエリがすでにエンコードされている場合や、一部の文字をエンコードせずに通過させる必要がある場合に便利です。

URL に必要な末尾の断片は、このメソッドを呼び出した後に手動で追加する必要があります。

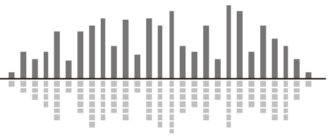


Typical use

```
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co"} ))
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co", Port = 49474} ))
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co", Path = "Path/with : and &"} ))
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co", Port = 49474, Path = "Path/with :
and &"} ))
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co", Path = "Path/with : and &", Query =
{ key = "headerValue", ["key with space"] = "header value with space"} } ))
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co", Port = 49474, Path = "Path with :
and &", Query = { key = "headerValue", ["key with space"] = "header value with space"} } ))
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co", Path = "Path/with : and &", Query =
{ key = "headerValue", ["key+space"] = "header+value+with+space"}, Encode = true } ))
print (HttpClient.CreateUrl( { Host = "http:/www.symetrix.co", Path = "Path/with : and &", Query =
{ key = "headerValue", ["key+space"] = "header+value+with+space"}, Encode = false } ))
```

このコードを実行すると、以下の返答が得られます。

```
http:/www.symetrix.co
http:/www.symetrix.co:49474
http:/www.symetrix.co/Path/with%20%3a%20and%20%26
http:/www.symetrix.co:49474/Path/with%20%3a%20and%20%26
http:/www.symetrix.co/Path/with%20%3a%20and%20%26?key=headerValue&key%20with%20space
=header%20value%20with%20space
http:/www.symetrix.co:49474/Path%20with%20%3a%20and%20%26?key=headerValue&key%20with
%20space=header%20value%20with%20space
http:/www.symetrix.co/Path/with%20%3a%20and%20%26?key=headerValue&key%2bspace=header
%2bvalue%2bwith%2bspace
http:/www.symetrix.co/Path/with%20%3a%20and%20%26?key=headerValue&key+space=header+v
alue+with+space
```



HttpClient.EncodeParams

EncodeParams メソッドは、渡されたパラメータのテーブルに対して、パラメータのフォーマットと URL エンコーディングを行います。これは、Download メソッドや Upload メソッドに渡す URL を生成する際に便利です。

EncodeParams メソッドは、エンコードされた Lua 文字列を正確に 1 つの値として返します。テーブル以外が渡された場合は、nil が返されます。返される値は、4095 文字を超えることはありません。入力文字列やエンコードされた文字列がこれを超える場合、文字列は切り捨てられます。

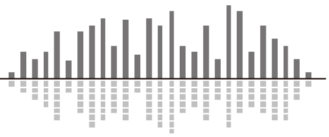
Parameters 引数は、1 つまたは複数のインデックス付きパラメータを含むテーブルである。各パラメーターは、インデックスと値で構成されています。インデックスと値は、書かれたとおりに使用されます。必要な"? "文字は、メソッドによって最初のクエリに付加され、"&"文字は、メソッドによって必要に応じて項目間に付加されます。空のクエリテーブルは、テーブルがないのと同じです。インデックスまたは値フィールドのスペースやその他の特殊文字は URL エンコードされ、スペースが「%20」に置き換えられるように、これらの文字を有効な URL 文字の同等物に置き換えます。各キーの最大長は 3823 文字です。長いキーは切り捨てられます。各値の最大長は、3823 文字です。長い値は切り捨てられます。キーと値のペアを追加すると、URL の最大長を超える場合は、追加されません。

例

```
output = HttpClient.EncodeParams( { key = "value", ["valid key"] = "valid value"} )
print(output)
```

このコードを実行すると、以下の返答が得られます。

```
key=value&valid%20key=valid%20value
```



HttpClient.EncodeString

EncodeString は、渡された文字列に対して URL エンコーディングを行います。スペースやその他の特殊文字は、有効な URL 文字の同等物に置き換えられます。これは、プログラマが Download および Upload メソッドで使用する URL 互換文字列を作成するために提供されるツールです。

EncodeString メソッドは、エンコードされた Lua 文字列を正確に 1 つの値として返します。エラーが発生した場合は、nil が返されます。

EncodeString メソッドは、以下の引数を使用します。

Arguments	Type	Required	Description
Input	string	Required	URL で無効なスペースやその他の文字を含む文字列
Encode Slash	bool	Optional	フォワードスラッシュをエンコードするかどうかを選択

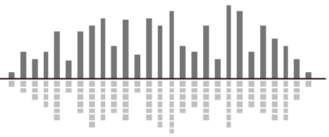
- **Input**
  入力された文字列の引数と返された値は 2047 文字を超えてはならず、これよりも長くなる場合は切り捨てられます。
- **Encode Slash**
  Encode Slash は、フォワードスラッシュ('/')をエンコードするか、そのままにするかを指定します。この引数はオプションで、存在しない場合はデフォルトで true が設定され、フォワードスラッシュがエンコードされます。

例

```
originalString = "Path/with : and &"
print(HttpClient.EncodeString(originalString))
print(HttpClient.EncodeString(originalString, false))
```

このコードを実行すると、以下の返答が得られます。

```
Path%2fwith%20%3a%20and%20%26
Path/with%20%3a%20and%20%26
```



HttpClient.DecodeString

DecodeString は、渡された文字列の URL デコードを行います。エンコードされたスペースやその他の特殊文字は、通常の読み取り可能な文字に置き換えられます。

これは、プログラマーが URL や Download メソッドから返されたコンテンツをデコードするために提供されるツールです。

DecodeString メソッドは、デコードされた Lua 文字列を正確に 1 つの値として返します。

文字列以外のものが渡された場合は、nil が返されます。

DecodeString メソッドは、次の引数を使用します。

Arguments	Type	Required	Description
Input	string	Required	エンコードされたスペースや、URL では無効なその他の文字を含む文字列。

- **Input**

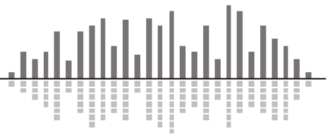
  Input 文字列引数および戻り値は 4095 文字を超えない。Input 引数または生成された文字列がこれより長い場合は、切り捨てられる。

Typical use

```
print(HttpClient.DecodeString("path%20with%20%3A%20and%20%26"))
```

このコードを実行すると、以下の返答が得られます。

```
Path with : and &
```

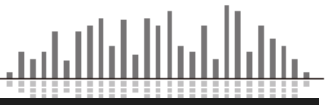


Usage Examples

Example 1： アップロードとダウンロードを同時に実行する

この例では、HttpClient.Upload と HttpClient.Download を同時に使用します

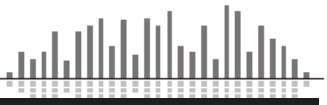
```
json = require("json")
--Post Test Server
--https://ptsv2.com/s/howitworks.html
ptsUrl = "https://ptsv2.com/t/a7nrl-1610735518/post"
--Note the Post Test Server in this example returns HTTP URL to the JSON version of
the upload so
that it can be subsequently downloaded. This was done by editing the "Body"
configuration option to
"{{URL-JSON}}".

-- Upload callback function
function UploadResponse(Table, ReturnCode, Data, Error, Headers)
     print("")
     print("Upload Callback:")
     print(string.format("URL requested = '%s'.", Table.Url))
     print(string.format("Data uploaded = '%s'.", Table.Data))
     print("Return Code: "..ReturnCode)
     print("Headers:")
     if (Headers ~= nil) then
          for headerName, headerValue in pairs(Headers) do
               print(string.format( "¥t"..headerName..": "..headerValue ) )
          end
     else
          print("¥tNone")
     end
     if (200 == ReturnCode) then
          print("Upload Success!")
          print(string.format("Data returned = '%s'", Data))
          --download the data just uploaded from the address provided by the return
data
          HttpClient.Download { Url = Data, Headers = { ["Content-type"] =
"text/*" }, Timeout = 3, EventHandler = DownloadResponse }
     else
          print(string.format("Failed, error code#%d, '%s'", ReturnCode, Error))
```



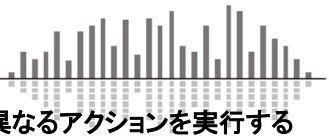
```
    end
end
-- Download callback function
function DownloadResponse(Table, ReturnCode, Data, Error, Headers)
    print("")
    print("Download Callback:")
    print(string.format("URL requested: '%s'.", Table.Url))
    print("Return Code: "..ReturnCode)
    print("Headers:")
    if (Headers ~= nil) then
        for headerName,headerValue in pairs(Headers) do
            print(string.format( "¥t"..headerName..": "..headerValue ) )
        end
    else
        print("¥tNone")
    end
    if (200 == ReturnCode) then
        print("Download Success!")
        print(string.format("Data returned = '%s'", Data))
        decodedData = json.decode(Data)
        print("Received: "..decodedData.Body)
    else
        print(string.format("Failed, error code#%d, '%s'", ReturnCode, Error))
    end
end

--Run the Example
HttpClient.Upload { Url = ptsUrl, Method = "POST", Data = "test message", Timeout =
3, EventHandler = UploadResponse }
```

このコードを実行すると、以下の返答が得られます。

```
Upload Callback:
URL requested = 'https://ptsv2.com/t/a7nrl-1610735518/post'.
Data uploaded = 'test message'.
Return Code: 200
Headers:
     Server: Google Frontend
     Access-Control-Allow-Origin: *
```



```
      Content-Length: 59
      Date: Tue, 26 Jan 2021 16:51:41 GMT
      Content-Type: text/plain; charset=utf-8
      Vary: Accept-Encoding
      X-Cloud-Trace-Context: aba8faf879dc03fdea1e77fd5237d104
Upload Success!
Data returned = 'http://ptsv2.com/t/a7nrl-1610735518/d/6483465671278592/json'
Download Callback:
URL requested: 'http://ptsv2.com/t/a7nrl-1610735518/d/6483465671278592/json'.
Return Code: 200
Headers:
      Server: Google Frontend
      Vary: Accept-Encoding
      Content-Length: 661
      Content-Type: application/json
      Date: Tue, 26 Jan 2021 16:51:41 GMT
      X-Cloud-Trace-Context: f282d7f7d4e548093d215da67afb8f4c
Download Success!
Data returned = '{"Timestamp":"2021-01-
26T16:51:41.287636Z","Method":"POST","RemoteAddr":"127.0.0.1:27812","ID":648346567127859
2,"Headers":
{"Accept":["*/*"],"Content-
Length":["12"],"Forwarded":["for=¥"2601:602:9600:2f8:212f:c94c:85a5:3738
¥";proto=https"],"Traceparent":["00-aba8faf879dc03fdea1e77fd5237d104-643f005180f011e6-
00"],"User-
Agent":["curl/7.55.1"],"X-Cloud-Trace-Context":
["aba8faf879dc03fdea1e77fd5237d104/7223492677381132774"],"X-Forwarded-For":
["2601:602:9600:2f8:212f:c94c:85a5:3738, 169.254.1.1"],"X-Forwarded-Proto":["https"],"X-
Google-Apps-
Metadata":["domain=gmail.com,host=ptsv2.com"]},"FormValues":{},"Body":"test
message","Files":null,"MultipartValues":null}'
Received: test message
```



Example 2 : 2つの異なる URL からデータをダウンロードし、返されたデータに基づいて異なるアクションを実行する

多くの場合、HttpClient.Download または HttpClient.Upload のソース URL に応じて、異なるアクションを実行する必要があります。

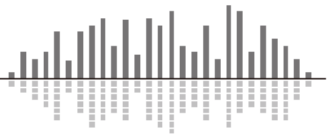
これは以下に示すように、異なるコールバック関数または単一のコールバック関数内の条件分岐で行うことができます。

```
url1 = "https://www.symetrix.co"
url2 = "https://duckduckgo.com"
-- callback function
function Response(Table, ReturnCode, Data, Error, Headers)
     print(string.format("URL requested = '%s'.", Table.Url))
    if (200 == ReturnCode) then
       print("Success!")
       if Table.Url == url1 then
          --do one thing
          print("Doing url1 thing")
       elseif Table.Url == url2 then
          --do another thing
          print("Doing url2 other thing")
       end
    else
          print(string.format("Failed, url %s, error code #%d, '%s'", Table.url,
ReturnCode,Error))
    end
end
HttpClient.Download { Url = url1, Headers = { ["Content-type"] = "text/*" }, Timeout
= 3,EventHandler = Response }
HttpClient.Download { Url = url2, Headers = { ["Content-Type"] = "text/*" }, Timeout
= 3,EventHandler = Response }
```

このコードを実行すると、以下の返答が得られます。

```
URL requested = 'https://www.symetrix.co'.
Success!
Doing url1 thing
URL requested = 'https://duckduckgo.com'.
Success!
Doing url2 other thing
```

この例では、さまざまな URL パスに簡単に適用できます。



Example 3 : REST 用文字変換機能

HTTP REST アプリケーションを扱う際に、文字列のテーブルを、各テーブル要素の間にフォワードスラッシュのセパレータを入れた単一のパス文字列に変換する必要がある場合があります。

これにより、指定子のリストから複雑なパスを作成することができます。

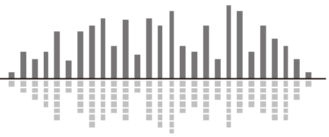
同様に、その連結された文字列をスラッシュで区切って、文字列のテーブルにすることが必要な場合もあります。このために 2 つの Lua ヘルパー関数が用意されています。

Concatenation（連結）

```
function createPathFromTable(inTable)
   if (type(inTable) == "table") then
       return(table.concat(inTable, "/"))
   else
       return("")
   end
end
```

Splitting（分割）

```
    function createTableFromPathString(inString)
   local outTable = {n = 0}
   if (type(inString) == "string") then
       local pattern = "(.-)" .. "/"
       local last = 1
       local head, tail, cap = inString:find(pattern, 1)
       while head do
            if head ~= 1 or cap ~= "" then
                 table.insert(outTable,cap)
            end
            last = tail + 1
            head, tail, cap = inString:find(pattern, last)
       end
       if last <= #inString then
            cap = inString:sub(last)
            table.insert(outTable, cap)
       end
   end
   return(outTable)
end
```

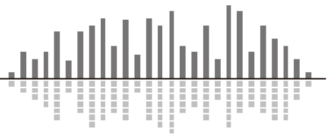


次のコードは、これらの関数の使用方法を示しています。

```
originalPathTable = {"dsp","channels",1,"output","fader"}
newPathString = createPathFromTable(originalPathTable)
print(newPathString)
newPathTable = createTableFromPathString(newPathString)
print(table.unpack(newPathTable))
```

このコードを実行すると、以下の返答が得られます。

```
dsp/channels/1/output/fader
dsp channels 1 output fader
```



JSON

Overview

JSON API を使用すると、JSON でエンコードされた文字列をエンコードおよびデコードできます（https://www.json.org/json-en.html）。 これは、一部のデバイスで制御プロトコルに使用されます。

Usage

使用するには、スクリプトに以下を追加する必要があります:

```
json = require('json'
```

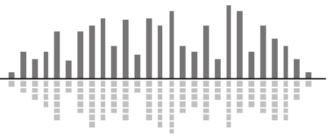
JSON

	Type	Comment
json.encode(object)	Function	object を JSON 文字列としてエンコードします
json.decode(jsonString, startPos)	Function	JSON 文字列をデコードし、Lua オブジェクトに返します
json.null()	Function	連想配列の null 値に使用されます

▪json.encode(object)

テーブル、文字列、ブール値、数値、nil、または json.null のオブジェクトを、JSON でエンコードした文字列を返します。

```
test =
{
name='Julie',
color='Red',
children={"Bob","Beth","Bruce"}
}

encodedString = json.encode(test)
print(encodedString)

-->>{"children":["Bob","Beth","Bruce"],"name":"Julie","color":"Red"}
```



▪json.decode(jsonString, startPos)

JSON 文字列をデコードし、デコードされた値を Lua データ 構造/値として返します。.

```
encodedString =
'{
"name":"Julie",
"color":"Red",
"children":["Bob", "Beth", "Bruce"]
}'

decoded = json.decode(encodedString)
```

オプションで、JSON 文字列が配置される開始位置を指定できます。 デフォルトは 1 です。.

```
encodedString =
'My JSON String: {
"name":"Julie",
"color":"Red","children":["Bob", "Beth", "Bruce"]
}'

decoded = json.decode(encodedString, 16)
```

スキャンされた JSON オブジェクトの後の最初の文字の位置である 2 番目の値が返されます。

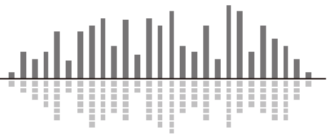
```
encodedString =
'My JSON String: {
"name":"Julie",
"color":"Red",
"children":["Bob","Beth","Bruce"]
} Something Else'

decoded, nextPosition = json.decode(encodedString, 16)
print(nextPosition)                          -->> 81
```

▪json.null()

この関数を使用すると、連想配列に null 値を指定できます（Lua で値を「nil」に設定した場合は破棄されます）。

```
t = { first=json.null() }
```

json.null または json.null() は同じ意味で使用される場合があることに注意してください。



Examples

次の例は、これらの使用方法を示しています。:

・Example 1 – エンコードとデコード

次の例では、テーブルをエンコードしてからデコードし、レビューのために結果を出力します。

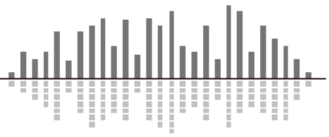
```
json = require('json')
test = {name='Julie',color='Red',children={"Bob","Beth","Bruce"}}
encodedString = json.encode(test)
print(encodedString)

decoded = json.decode(encodedString)

for i, k in pairs(decoded) do
    if type(k) ~= "table" then
        print(i, k)
    else
        print(i)
        for m, n in pairs(k) do
            print("  ", m, n)
        end
    end
end
```

デバッグ出力ファイルに次のような出力が表示されます。:

```
{"name":"Julie","color":"Red","children":["Bob","Beth","Bruce"]}
name  Julie
color Red
children
      1     Bob
      2     Beth
      3     Bruce
```

Lua テーブルには順序がないため、返されるテーブルアイテムの順序が異なる場合があることに注意してください。

上記は、多次元テーブルを印刷するための良い例も示しています。



・Example 2 – nil vs json.null

上記のように、json.null を使用すると、連想配列に null 値を指定できます。

Lua で値を「nil」に設定した場合、これは破棄されます。

これを使用すると、以下の例に示すように、JSON でエンコードされたときにキーを保持できます。

```
user1 = {firstName="Jack", lastName=nil}
user2 = {firstName="Jill", lastName=json.null}

encodedUser1 = json.encode(user1)
encodedUser2 = json.encode(user2)

print("encoded user1: " .. encodedUser1)
print("encoded user1: " .. encodedUser2)
-->> encoded user1: {"firstName":"Jack"}
-->> encoded user1: {"firstName":"Jill","lastName":null}
```

user2 はキーを保持し、JSON エンコーディングで「null」に設定されているが、user1 のキーは破棄されています。

NamedControl

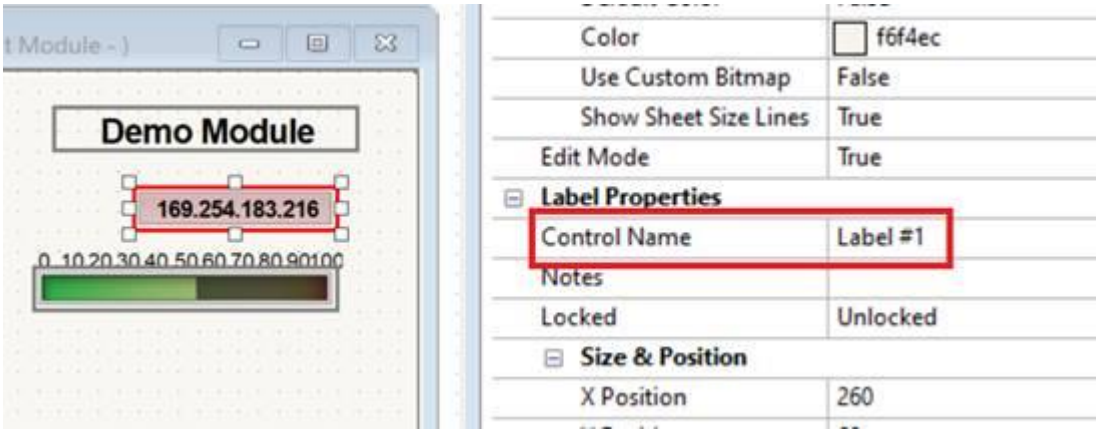
Overview

インテリジェントモジュール UI に追加されたコントロールとその基になる DSP パラメータには、NamedControl インターフェイスを使用してスクリプトからアクセスできます。
次に、Composer の広範なリモートコントロールシステムを使用して、インテリジェントモジュール UI の NamedControl を、リモートコントロール番号を使用してシステム内の他のコントロールにリンクできます。

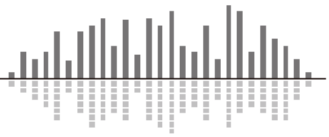
NamedControl.

	Return Type	Return Range	Comment
NamedControl.GetText(name)	String	String	コントロールのテキストを取得
NamedControl.SetText(name, value)	Function		コントロールのテキストを設定
NamedControl.GetValue(name)	Float	User adjustable range	コントロールの値を取得
NamedControl.SetValue(name, value)	Function		コントロールの値を設定
NamedControl.GetPosition(name)	Float	0–1	コントロールのポジションを取得
NamedControl.SetPosition(name, position)	Function	0–1	コントロールのポジションを設定

API は、編集可能なコントロール”Name”を各関数の引数として使用します。
Name は、各コントロールの編集可能なプロパティ「Control Name」に対応しています。



Control Name は唯一にし、重複しないように注意してください。
同じ名前のコントロールが複数ある場合、スクリプトは最初に配置されたコントロールを選択します。
コントロールのデフォルト名は一意ですが、コントロールをコピーして貼り付けると名前が重複します。
これらのコントロールプロパティの詳細については、インテリジェントモジュールコントロールビューレイアウトを参照してください。
NamedControl API は、コントロールタイプごとに動作が異なります。 ここでは、制御タイプについて詳しく説明します。



・NamedControl.GetText(name)

指定された名前のコントロールに表示される、テキストの文字列を返します。

```
Print(NamedControl.GetText("myLabel"))          -->> Ben
```

これは、次のコントロールタイプで機能します:

Control	Function
Label	ラベルテキストを取得
Numeric	単位と丸めを含む、数値表示テキストを取得
Fader	単位と丸めを含むテキストとして、フェーダー値を取得
Button	n/a
Radio Button	n/a
LED	n/a
Multistate LED	n/a
Meter	n/a
Gauge	n/a
Drop List	n/a
Video Stream	表示されている画像の URL を取得

・NamedControl.SetText(name, value)

指定されたテキストを使用して、指定された名前のコントロールに表示される、テキストを設定します。

```
NamedControl.SetText("myLabel", "Ben")          -- Set Label to Ben
```

これは、次のコントロールタイプで機能します:

Control	Function
Label	ラベルテキストを設定
Numeric	n/a
Fader	n/a
Button	n/a
Radio Button	n/a
LED	n/a
Multistate LED	n/a
Meter	n/a
Gauge	n/a
Drop List	n/a
Video Stream	表示する画像の URL を設定

ラベルとビデオストリームの場合は.SetText 関数を使用し、他のすべてのコントロールタイプの場合は、以下で説明する.SetValue または.SetPosition を使用することに注意してください。

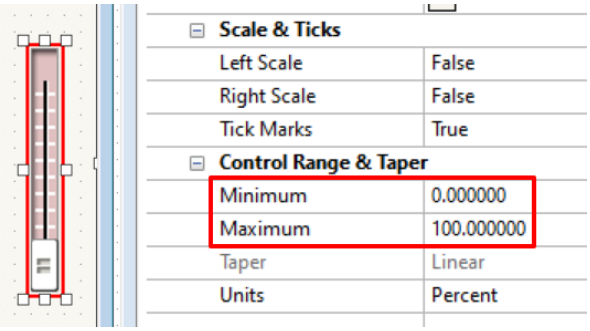
すべてのコントロールは、プロパティパネルを使用してテキストを静的な値に設定できます。

・NamedControl.GetValue(name)

指定された名前のコントロールの、フロート値を返します。

```
Print(NamedControl.GetValue("myFader"))          -->> 40
```

返されるフロート値は、コントロールの Properties パネルの Minimum および Maximum 項目で指定された範囲内です。 Minimum とは、最小制御位置での値を指します。 同様に、Maximum は最大制御位置での値です。 コントロールを逆方向に動作させたい場合は、値を反転できます。 これらの制限間の分布は、Taper プロパティが使用されます。



これは、次のコントロールタイプで機能します:

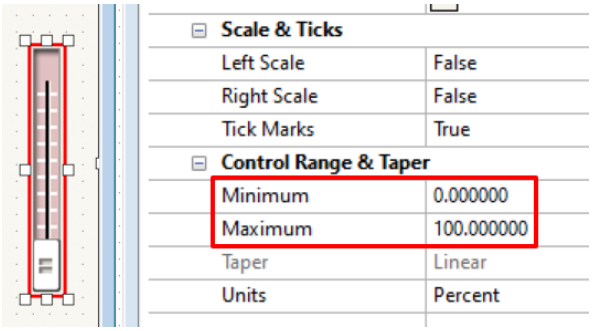
Control	Function
Label	n/a
Numeric	数値の値を取得 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Fader	フェーダーの値を取得 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Button	ボタンの値を取得 0 = disabled (OFF), 1 = enabled (ON)
Radio Button	選択したラジオボタンの値を取得 値は 0 インデックスによる、選択したボタンのフロート値 (例えば グループの 8 番目のボタンは 7.0)
LED	LED の値を取得 0 = disabled (OFF), 1 = enabled (ON)
Multistate LED	マルチステート LED の値を取得 値は LED の状態に基づいて、0 と 1 の間で均等に分散されます
Meter	メーターの値を取得 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Gauge	ゲージの値を取得 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Drop List	選択したドロップリストアイテムの値を取得 値は 0 インデックスによる、選択したアイテムのフロート値 (例えば ドロップリストの 8 番目のアイテムは 7.0)
Video Stream	n/a

・NamedControl.SetValue(name, value)

指定された名前のコントロールの値を、指定された値に設定します。

```
NamedControl.SetValue("myFader", 50)      -- Fader value set to 50
```

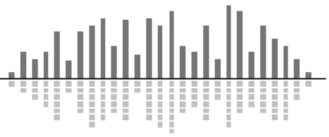
返されるフロート値は、コントロールの Properties パネルの Minimum および Maximum 項目で指定された範囲内です。 Minimum とは、最小制御位置での値を指します。 同様に、Maximum は最大制御位置での値です。 コントロールを逆方向に動作させたい場合は、値を反転できます。 これらの制限間の分布は、Taper プロパティが使用されます。



これは、次のコントロールタイプで機能します:

Control	Function
Label	n/a
Numeric	数値の値を設定 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Fader	フェーダーの値を設定 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Button	ボタンの値を設定 0 = disabled (OFF), 1 = enabled (ON)
Radio Button	選択したラジオボタンの値を設定 値は 0 インデックスによる、選択したボタンのフロート値 (例えば グループの 8 番目のボタンは 7.0)
LED	LED の値を設定 0 = disabled (OFF), 1 = enabled (ON)
Multistate LED	マルチステート LED の値を設定 値は LED の状態に基づいて、0 と 1 の間で均等に分散されます
Meter	メーターの値を設定 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Gauge	ゲージの値を設定 範囲とテーパーは、コントロールプロパティでユーザーが構成できます
Drop List	選択したドロップリストアイテムの値を設定 値は 0 インデックスによる、選択したアイテムのフロート値 (例えば ドロップリストの 8 番目のアイテムは 7.0)
Video Stream	n/a

ほとんどのコントロールは.SetValue 関数を使用しますが、上記で説明するように、ラベルとビデオストリームコントロールは.SetText を使用することに注意してください。



・NamedControl.GetPosition(name)

指定された名前のコントロールの”位置・ポジション”のフロート値を返します。

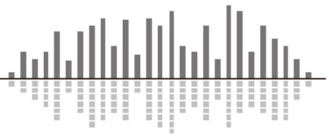
```
Print(NamedControl.GetPosition("myFader"))          -->> 0.4
```

ポジションは値(Value)に似ていますが、常に 0-1 の間にあり、テーパーとは無関係です。

言い換えると、値は制御範囲とテーパーの影響を受けますが、ポジションは影響を受けません。

これは、次のコントロールタイプで機能します:

Control	Function
Label	n/a
Numeric	数値の位置を取得 範囲は、直線的に広がる 0 から 1 の間
Fader	フェーダーの位置を取得 範囲は、直線的に広がる 0 から 1 の間
Button	ボタンの位置を取得 0 = disabled(OFF), 1 = enabled(ON)
Radio Button	選択したラジオボタンの位置を取得 位置は、ラジオボタンに基づいて 0 と 1 の間で均等に分散されます
LED	LED の位置を取得 0 = disabled(OFF), 1 = enabled(ON)
Multistate LED	マルチステート LED の位置を取得 位置は、LED の状態に基づいて 0 と 1 の間で均等に分散されます
Meter	メーターの位置を取得 範囲とテーパーは、コントロールプロパティでユーザーが構成できます。
Gauge	ゲージの位置を取得 範囲とテーパーは、コントロールプロパティでユーザーが構成できます。
Drop List	選択したドロップリストアイテムの位置を取得 ドロップリストの項目に基づいて、位置は 0 と 1 の間で均等に分散されます。
Video Stream	n/a



・NamedControl.SetPosition(name, position)

指定された名前のコントロールの位置を、指定された値に設定します。

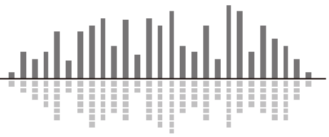
```
NamedControl.SetPosition("myFader", .5)   -- Fader position set to .5
```

繰り返しますが、ポジションは値(Value)に似ていますが、常に 0–1 の間にあり、テーパーとは無関係です。

言い換えると、値は制御範囲とテーパーの影響を受けますが、ポジションは影響を受けません。

これは、次のコントロールタイプで機能します:

Control	Function
Label	n/a
Numeric	数値の位置を設定 範囲は、直線的に広がる 0 から 1 の間
Fader	フェーダーの位置を設定 範囲は、直線的に広がる 0 から 1 の間
Button	ボタンの位置を設定 0 = disabled(OFF), 1 = enabled(ON)
Radio Button	選択したラジオボタンの位置を設定 位置は、ラジオボタンに基づいて 0 と 1 の間で均等に分散されます
LED	LED の位置を設定 0 = disabled(OFF), 1 = enabled(ON)
Multistate LED	マルチステート LED の位置を設定 位置は、LED の状態に基づいて 0 と 1 の間で均等に分散されます
Meter	メーターの位置を設定 範囲とテーパーは、コントロールプロパティでユーザーが構成できます。
Gauge	ゲージの位置を設定 範囲とテーパーは、コントロールプロパティでユーザーが構成できます。
Drop List	選択したドロップリストアイテムの位置を設定 ドロップリストの項目に基づいて、位置は 0 と 1 の間で均等に分散されます。
Video Stream	n/a



Usage Examples

次の例は、これらの使用方法を示しています:

・Example 1 – Fader の値を読み取る

NamedControl 関数を使用する一般的な方法は、タイマーを使用してそれらをポーリングすることです。一定間隔ごとに入力の値が読み取られ、いくつかの処理が実行され、UI が更新されます。

```
function TimerClick ()
   newValue = NamedControl.GetValue("MyFader")
   -- Do some processing
   newValue = newValue * 0.8
NamedControl.SetValue("MyMeter", newValue)
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(0.5)                  -- Update the UI every 0.5 sec
```

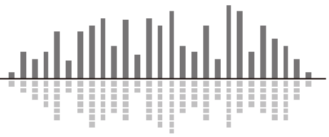
・Example 2 – 変化があった時のみ Fader の値を読み取る

これらの TimerClick()関数の処理は、可能な限り効率的にすることが重要です。定期的に更新されない UI 処理は、個別に確認して処理する必要があります。また、画面上のコントロールが変更された場合にのみ更新する必要があります。ほとんどの場合、コントロールの現在の状態を保存し、次のタイマークリック時にコントロールの更新された状態と比較する必要があります。 その後、新しい値に基づいて何らかのアクションを実行できます。

変更されていない場合は、一連の作業をスキップできます。 したがって、例 1 は次のように書き直すことができます。

```
currentValue = 0
function TimerClick ()
      newValue = NamedControl.GetValue("MyFader")
      if newValue ~= currentValue then    --Compare Values
         -- Do some processing
         newValue * 0.8
         NamedControl.SetValue("MyMeter", newValue)
         currentValue = newValue --Update for the next pass
      end
end
MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(0.5)
```

これはより多くのコードを使用しますが、必要な場合にのみ UI の処理と更新を行うため、より効率的です。



・Example 3 – セルフクリアラッチボタン

ほとんどの場合、モーメンタリボタンよりもセルフクリアラッチボタンを使用することをお勧めします。

これは、特にタイマー間隔が遅い場合、瞬間的なボタンへの変更が見落とされたり、誤って解釈されたりする可能性があるためです。

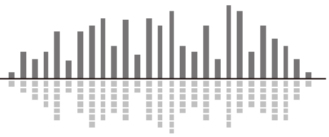
たとえば、ある TimerClick 処理の後で、次の TimerClick 処理の前に瞬間的にボタンを押して離した場合、そのアクションは失われます。

サポートされている最速の 0.25 秒の間隔でも、これは発生する可能性があります。

また、これらをキャッチするためだけにタイマーを高速で実行する必要はありません。これは他の方法で必要になるよりもはるかに多くの CPU を消費するためです。

ボタンが押されたことを見逃さないようにするには、値が読み取られた後に状態をクリアするコードを含むラッチボタンを使用するのが最善です。

```lua
function TimerClick ()
   newValue = NamedControl.GetValue("MyButton")
   if newValue == 1 then   --Check if pressed
      -- Do some stuff when the button is pressed
      --Reset the button by turning it off
      NamedControl.SetValue("MyButton", 0)
   end
end
MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(0.5)
```



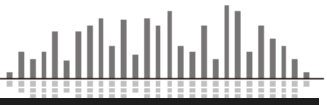
・Example 4 – 長押しボタン

モーメンタリボタンの良い使い方の 1 つは、長時間押したときに機能が異なるボタンを実装することです。

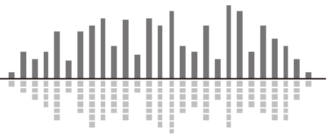
たとえば、ボタンを 5 秒間押したときに 2 とは異なる機能があり、ボタンの押下が受け入れられたことをユーザーに個別にフィードバックするメカニズムがあります。

これには、許容可能な最短のプレスがタイマー間隔よりも長いことが必要です。

```
--Button Times
resultOneTime = 2000
resultTwoTime = 5000 --should be longer than resultOneTime

--State Variables
startTime = System.GetTime()
timeElapsed = 0
resultOneComplete = false
resultTwoComplete = false
runInProgress = false

function TimerClick ()
    newButtonValue = NamedControl.GetValue("ButtonAction")
    --Check if run in progress and button is pressed
    if runInProgress == false and newButtonValue == 1 then
        --Clear Results
        resultOneComplete = false
        resultTwoComplete = false
        --start measuring time from now
        startTime = System.GetTime()
        timeElapsed = 0
        --start run
        runInProgress = true
    elseif runInProgress == true and newButtonValue == 1 then
        timeElapsed = System.GetTime() - startTime
        if timeElapsed >= resultTwoTime then
            --Do work when Time Two is reached
            resultTwoComplete = 1
        end
    elseif runInProgress == true and newButtonValue == 0 then
        if timeElapsed >= resultOneTime and timeElapsed < resultTwoTime then
            --Do work when Time One is reached and button is released
            resultOneComplete = 1
```



```
        end
        --end run
        runInProgress = false
        --stop measuring time
        timeElapsed = 0
    end
    --update the UI
end
MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(.25)
```



SSH

Overview

Ssh API は、SSH で他の機器と通信することができます。

SSH API は、コントロールネットワークを使用してのみ動作します。

SSH API は、TcpSocket API と多くの類似点があります。

SSH.

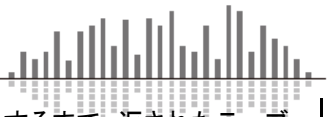
	Return Type	Comment
Ssh.New()	SSH object	新しい SSH オブジェクトを作成するために使用されます。

New() は、グローバルテーブルとして Ssh オブジェクトを作成します。

```
MySsh = Ssh.New()
```

SshName

	Return Type	Comment
SshName.ID	Integer	スクリプト内のソケットの番号。
SshName.EventHandler(, event, error)	Callback	Ssh テーブルの引数、Event テーブルのイベント、任意のエラー文字列でイベントが発生した場合に呼び出される。
SshName.ReconnectTimeout	Number	ソケットが外部から切断された場合、再接続するまでの待ち時間(秒)。
SshName.IsConnected	Boolean	ソケットが現在接続中であるかどうかを示す。
SshName.IsInteractive	Boolean	接続に疑似端末が必要な場合は true を設定する。
SshName.BufferLength	Integer	受信したバイト数および読み込み待ちのバイト数。
SshName.PublicKey	String	SSH 形式で指定された公開鍵(必要な場合)。
SshName.PrivateKey	String	秘密鍵(必要に応じて、PEM 形式で指定します)。
SshName.PrivateKeyPassword	String	秘密鍵のパスワード(必要な場合)。
SshName:Connect(ip, port, username, password)	Method	ソケットを IP アドレスに接続するために呼び出される。
SshName:Disconnect()	Method	接続されたソケットを切断する。
SshName:Write(data)	Method	接続されたソケットにデータのテーブルを書き込む。
SshName:Read(length)	Method	接続されたソケットから返されたテーブルに整数バイトのカウントを読み込む。



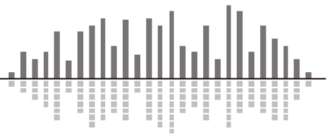
SshName:ReadLine(EOL, [delimiter])	Method	指定された EOL 列挙ケースに遭遇するまで、返されたテーブルに読み込む。 カスタムデリミタの場合は、文字列で指定する。
SshName:Search(pattern, [start])	Method	ソケット内の未読データを検索して、オプションの 1 ベースの開始インデックスから始まる文字列を検索する。1 ベースのインデックスを返すが、データは返さない。
SshName.LoginFailed(, error)	Callback	Ssh テーブルとエラー文字列を引数に持つ接続が、ログインに失敗したときに呼び出される。
SshName.Connected( )	Callback	ソケットが Ssh テーブルの引数で接続されたときに呼び出される。
SshName.Reconnect( )	Callback	Ssh テーブルの引数で、ソケットが再接続を試みているときに呼び出される。
SshName.Data(, data)	Callback	Ssh テーブルとデータテーブルを引数として、未読のデータがあるときに呼び出される。
SshName.Closed( )	Callback	Ssh テーブルの引数で、ソケットがクローズされたときに呼び出される。
SshName.Error(, error)	Callback	Ssh テーブルの引数とエラー文字列で、エラーが発生したときに呼び出される。
SshName.Timeout(, error)	Callback	TcpScocket テーブルの引数とエラー文字列で、読み取りまたは書き込みのタイムアウト時に呼び出される。

▪ SshName.ID

スクリプト内のソケットの 0～9 の番号です。

```
print(MySsh.ID)                    -->> 1
```

最大 8 つの Ssh オブジェクトを別々の呼び出しで同時に要求することができます。ソケットがクローズされると（下記参照）、別の SshName.New() メソッド呼び出しで再び使用できるようになります。



・SshName.EventHandler(Ssh,event,error)

Ssh テーブルの引数、イベントテーブルのイベント、および任意のエラー文字列でイベントが発生するたびに呼び出される、Ssh のコールバック関数を割り当てます。

これは、2 つの方法で行うことができます。専用の関数を使用する:

```
function SshHandler (sock, evt, err)
   --handle the event
end

SshName.EventHandler = SshHandler
```

または、匿名関数を使用することもできます

```
SshName.EventHandler = function(sock, evt, err)
   --handle the event
end
```

通常、if-elseif-else ステートメントで複数の可能なイベントタイプを処理することになります。比較の可読性を高めるために、"Ssh.Events."の列挙の表があります。

Enumeration	Value	Functionality
"Connected"	1	Ssh が接続されました。
"Reconnect"	2	Ssh は再接続を試みています。
"Data"	3	Ssh はデータが利用可能です。
"Closed"	4	Ssh が終了しました。
"Error"	5	Ssh がエラーになりました。
"Timeout"	6	Ssh がタイムアウトしました。
"LoginFailed"	7	SSH ソケットにログイン失敗が発生しました。

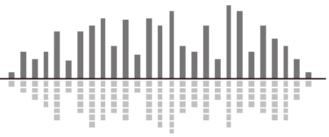
・SshName.ReconnectTimeOut

Ssh ソケットの ReconnectTimeout を設定し、デフォルト値である 5 秒を上書きします。これは、ソケットが外部から切断された場合に、再接続を試みるまでの待ち時間を秒単位で設定するものです。再接続を試みると、Reconnect コールバックが呼び出されます。成功した場合は、Connected コールバックが呼び出される。0 を設定すると、再接続を試みないようになります。

```
SshName.ReconnectTimeout = 1
```

通常はデフォルトで問題ありませんが、接続先のデバイスが再接続までに長い時間を必要とすることが分かっている場合は、これを調整する必要があるかもしれません。

さらに、接続状態を手動で監視し、コード内に再接続のパスを用意しておくのも良い方法です。



・SshName.IsConnected

ソケットが現在接続されているかどうかを示す。

```
print(SshName.IsConnected) -->> True
```

接続状態をテストし、適切な処置を行うために使用されます。

・SshName.IsInteractive

IsInteractive プロパティは、SSH サーバーに PTY モード用の通信フォーマットを指示するために使用されます。 このプロパティが ″true ″に設定されている場合、ベーシックな“vanilla”ターミナルエミュレーションが使用されます。デフォルトでは、″false ″に設定されています。

```
SshName.IsInteractive = true
```

デバイスやサービスによっては、このプロパティを特定の方法で設定する必要があります。スクリプトの接続に問題があるが、他の設定がすべて正しく見える場合は、このプロパティを ″true ″に設定してみてください。

・SshName.BufferLength

Buffer から受信し、読み取りを待っているバイトのカウントです。

```
print(SshName.BufferLength) -->> 8
```

Read コールと組み合わせて使用することが多い。実際の使用方法については、下記と例 1 を参照してください。

```
rxLine = SshName:Read(ssh.BufferLength)
```

BufferLength プロパティは、Read や ReadLine メソッド呼び出しなど、その値を変更する可能性のある操作のたびに更新され、Buffer からデータが削除されます。新しいデータが到着するとバッファに追加されますが、これはスクリプトの実行中にのみ発生するため、スクリプトの実行中に Read Buffer と BufferLength プロパティの値が増加することはありません。

・SshName.PublicKey

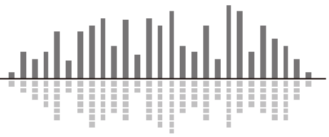
PKI 認証を使用する場合は、プロパティに SSH 形式の公開鍵文字列を設定します。

```
SshName.PublicKey =
"<encryption algorithm> <key> <optional comment>"
```

典型的な暗号化アルゴリズムは「ssh-rsa」なので、例えば PublicKey プロパティは以下のように設定します：

```
SshName.PublicKey =
"ssh-rsa AAAAB3NzaC2yc2EAAAADAi0657Y2YoSG+CdR1h1EQa1fcYX50y8P
Comment-about-the-key"
```

公開鍵の最大長は ASCII 文字で 4095 文字です



・SshName.PrivateKey

PKI 認証を使用する場合は、このプロパティに PEM 形式（OpenSSL）の秘密鍵文字列を設定します。この文字列は非常に長いため、文字列を入力するための引用符なしの複数行形式が使用されます。

```
SshName.PrivateKey = [[-----BEGIN OPENSSH PRIVATE KEY-----
<key>
 -----END OPENSSH PRIVATE KEY-----]]
```

そのため、例えば、PrivateKey プロパティには、以下のように設定することができます：

```
SshName.PrivateKey = [[-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABFwAAAAdz
...
7euEFqyIRfs+fGQC/LefbBrFugjPPmkAAAAAAQIDBAUGBwgJCg5=
-----END OPENSSH PRIVATE KEY-----]]
```

秘密鍵の最大長は、ASCII 文字で 4095 文字です。

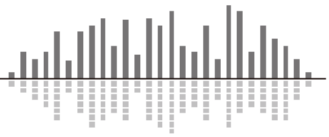
・SshName.PrivateKeyPassword

PKI 認証を利用する場合、秘密鍵にパスワードが必要な場合があります。

```
SshName.PrivateKeyPassword = "My-unbreakable-password"
```

秘密鍵パスワードの最大長は、ASCII 文字で 255 文字です。

秘密鍵パスワードが不要な場合は、このプロパティを空白にしてください。



・SshName.Connect(ip,port,username,password)

与えられたユーザー名とパスワードで、IP アドレスまたはホスト名とポート(0-65535)にソケットを接続するために呼び出されます。ユーザー名とパスワードは ASCII であることが前提です。どちらも最大文字数は 255 文字です。

```
SshName:Connect("169.254.179.214", port, "username", "password")
```

公開鍵/秘密鍵のペアを使用する場合は、「パスワード」に空文字列を設定します。

```
SshName:Connect("169.254.179.214", port, "username", "")
```

Device の情報を使って、正しい IP アドレスが使用されていることを確認することができます。

```
currentDeviceIP = Device.RemoteUnit.DanteIP
sock:Connect(currentDeviceIP, port, "username", "password")
```

認証が成功した場合、Connected コールバックと EventHandler コールバックが呼び出されます。

失敗した場合は、LoginFailed コールバックと EventHandler コールバックが呼び出されます(使用される場合)。

接続されると、スクリプトアプリは SSH 接続から新しいデータがないかポーリングします。データが存在する場合、データは最大バッファサイズである 64K バイトまでバッファリングされます。Read メソッドと ReadLine メソッドが呼び出されると、データはバッファからセカンダリバッファにコピーされ、LUA スクリプトに戻されます。これは、スクリプトによって読み込まれると、一次バッファは直ちに新しいデータのためのスペースを確保しなければならないからである。また、リターン・バッファは最大 64K バイトである。これより大きな送信の場合、新しいデータが失われることはありませんが、ReadLine/Search コールがあっても検索文字が見つからず、スクリプトがデータを読み込むまでデータ受信が滞ってしまいます。

これは、Read や ReadLine メソッドが呼び出されるまで受信データを実際にバッファリングしない TcpSocket API とは異なることに注意してください。TcpSocket では、ReadLine メソッドは、必要な EOL または検索文字が見つからない限り、データをバッファリングしません。

・SshName.Disconnect()

接続されているソケットを切断するために呼び出します

```
SshName:Disconnect()                    -- socket is disconnected
```

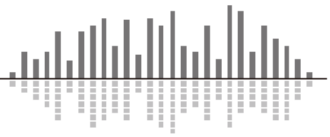
・SshName.Write(data)

接続されたソケットにデータのテーブルを書き込みます。

```
SshName:Write('V¥x0d')                  -- Data is written and sent
```

接続されると、Lua スクリプトは接続先にデータを書き込むことができます。

書き込みは 64K を超えないようにしてください。64K を超えるとエラーが返され、書き込みは中止されます。



・SshName.Read(length)

接続されたソケットバッファから、整数 "length "のバイトを返されたテーブルに読み込みます。

```
rx = SshName:Read(10000)
```

読み取ったバイトはバッファから削除されます。

・SshName.ReadLine(EOL,[delimiter])

指定した EOL が見つかるまで返されたテーブルを読み込みます。

```
rxLine = Ssh:ReadLine(1)
--Read until any combination of linefeeds and returns are reached
```

指定できる EOL は下表のとおりです。

Enumeration	Value	Functionality
Any	1	CR と LF の任意の組み合わせを検索
CrLf	2	CR または LF を検索
CrLfStrict	3	CR+LF を検索
Lf	4	LF を検索
Null	5	Null(0x00)を検索
Custom	6	カスタム文字列を検索

```
rxLine = SshName:ReadLine(Ssh.EOL.Any)
--Read until any combination of linefeeds and returns are reached
```

EOL 引数として'Custom'が渡された場合、第 2 引数として文字列デリミターを含める必要がある。

```
rxLine = SshName:Readline(6, "TheEnd")
--Read until "TheEnd" is reached
```

・SshName.Search(pattern,[start])

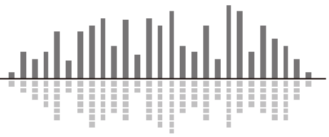
ソケット内の未読 Buffer から、オプションの 1 ベースの "start "インデックスで始まる文字列を検索する。

"start"インデックスが与えられない場合、検索は Buffer の先頭から開始される。Buffer を変更することなく、検索パターンの最初の文字の 1 ベースのインデックスを返す。

```
SshName:Search("ImportantMessage", 1)              -->> 16
SshName:Search("AnotherMessage", 17)               -->> nil
```

検索は、渡された文字列を正確に探します。検索文字列の特定の文字(例:*や?)に対する解釈はなく、Lua のパターンマッチはサポートされていません。

検索は、特定のレスポンスを探しているような状況で効率を上げるために使用できます。これは、多くのデータを送信するプロトコルで、1 つのことにしか関心がない場合に便利です。各行を読んでデータを解析し、特定の文字列を探す代わりに、スクリプトはそれを検索し、それが存在しない場合は、1 回の Read()コマンドで受信したデータをすべて捨ててしまうことができます。



・SshName.LoginFailed(Ssh,error)

接続中にログインに失敗したときに呼び出されるコールバック関数を定義します。このエラーは有用な情報を提供するので、通常はデバッグに出力されるか、ユーザーに提示されます。

これには、2 つの方法があります。

```
function LoginFailedHandler (Ssh, error)
   --handle the failure
   print(error)
end

SshName.LoginFailed = LoginFailedHandler
```

または、匿名関数を使用することもできます

```
SshName.Connected = function(Ssh)
   --handle the new Connection
end
```

・SshName.Connected(Ssh)

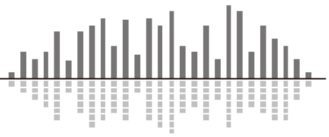
Ssh テーブルの引数でソケットが接続されたときに呼び出されるコールバック関数を定義します。

これには、2 つの方法があります。

```
function ReconnectHandler (Ssh)
   --handle the Reconnection attempt
end

SshName.Reconnect = ReconnectHandler
```

または、匿名関数を使用することもできます

```
SshName.Connected = function(Ssh)
   --handle the new Connection
end
```



・SshName.Data(Ssh,data)

Ssh テーブルとデータテーブルを引数として、未読データがある場合に呼び出されるコールバック関数を定義します。

これには、2 つの方法があります。

```
function DataHandler (Ssh, data)
   --handle the data
end

SshName.Data = DataHandler
```

または、匿名関数を使用することもできます

```
SshName.Data = function(Ssh, data)
   --handle the data
end
```

データは、Read 関数や ReadLine 関数で読み込むことができます。

これは、一般的な EventHandler コールバックの代わりに使用することができます。

・SshName.Closed(Ssh)

Ssh テーブルの引数でソケットを閉じたときに呼び出されるコールバック関数を定義します。

一度接続した後、Disconnect メソッドを明示的に呼び出す以外の理由で切断された場合、Closed コールバックが呼び出されます。再接続タイムアウトが 0 でない場合、アプリはプログラムされた時間後に再接続を試みます。

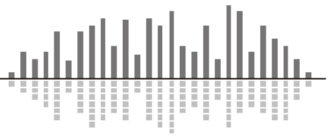
これには、2 つの方法があります。

```
function ClosedHandler (Ssh)
   --handle the socket closing
end

SshName.Closed = ClosedHandler
```

または、匿名関数を使用することもできます

```
SshName.Closed = function(Ssh)
   --handle the socket closing
end
```

これは、一般的な EventHandler コールバックの代わりに使用することができます。



・SshName.Error(Ssh,error)

Ssh テーブルとエラー文字列を引数に、エラーが発生したときに呼び出されるコールバック関数を定義します。

エラー文字列は問題を診断するのに便利なので、デバッグログに出力するなどしてユーザーに提示することがよくあります。

これには、2 つの方法があります。

```
function ErrorHandler (Ssh, error)
   --handle the error
end

SshName.Error = ErrorHandler
```

または、匿名関数を使用することもできます

```
SshName.Error = function(Ssh, error)
   --handle the error
end
```

これは、一般的な EventHandler コールバックの代わりに使用することができます。

・SshName.Timeout(Ssh,error)

読み取りまたは書き込みのタイムアウト時に呼び出されるコールバック関数を定義します。

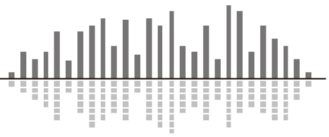
これには、2 つの方法があります。

```
function TimeoutHandler (Ssh, error)
   --handle the Timeout
end

SshName.Timeout = TimeoutHandler
```

または、匿名関数を使用することもできます

```
SshName.Timeout = function(Ssh, error)
   --handle the Timeout
end
```

これは、一般的な EventHandler コールバックの代わりに使用することができます。



Usage Examples

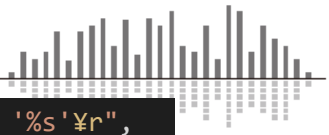
次の例は、これらの使用方法を示しています:

・Example 1 – パスワードを使用して SSH 接続する

この例では、Ssh がシンプルなパスワード接続で一般的に使用される方法を示します。

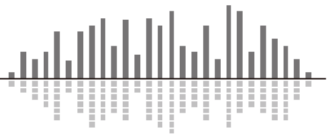
この例では、1 つの EventHandler を使用しています。

```
SshName = Ssh.New()
address = "server.address" --enter correct address
port = 22 --enter correct port number
user = "user_name" --enter correct user name
password = "password" --enter correct password

--EventHandler
SshName.EventHandler = function(Ssh, evt, error)
    if evt == Ssh.Events.Connected then
        --handle the new Connection
        print("socket connected¥r")
        --begin sending data using ssh:Write as needed
    elseif evt == Ssh.Events.Reconnect then
        --handle the Reconnection attempt
        print("socket reconnecting...¥r")
    elseif evt == Ssh.Events.Data then
        --handle the data
        rxLine = sock:Read(ssh.BufferLength)
        if (nil ~= rxLine) then
            print(rxLine)
        end
    elseif evt == Ssh.Events.Closed then
        --handle the socket closing
        print("socket closed by remote¥r")
    elseif evt == Ssh.Events.Error then
        --handle the error
        print(string.format("Error: '%s'¥r", error))
    elseif evt == Ssh.Events.Timeout then
        --handle the Timeout
        print("socket closed due to timeout¥r")
    elseif evt == Ssh.Events.LoginFailed then
```



```
        print("login failed with " .. string.format("error: '%s'¥r",
error))
    else
        print("unknown socket event¥r")
  end
end

SshName:Connect(address, port, user, password)
```

受信した ″evt ″を比較し、適切なアクションを実行するために、上記の Event 列挙テーブルが使用されます。一般的に常に列挙表にある 6 種類の可能なイベントタイプを全て処理する必要があります。

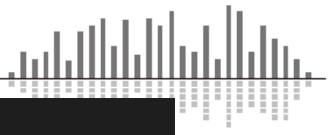


・Example 2 – ディスクリートハンドラーを使った鍵付き SSH 接続

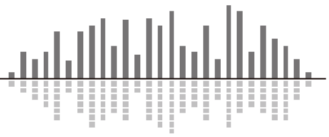
すべてのイベントを 1 つのハンドラで処理する代わりに、イベントの種類ごとに別のハンドラを使用することもできます。この方が、if-ifelse の大きなセクションが必要ないので、読みやすくなる傾向があり、好ましいかもしれません。

前回と同様に、一般的には 7 つのイベントすべてを処理する必要があります。

```
SshName = Ssh.New()
address = "server.address" --enter correct address
port = 22 --enter correct port number
user = "user_name" --enter correct user name
password = "" --for key connection, password should be blank

-- public key in SSH format
SshName.PublicKey = "ssh-rsa Your_Public_Key"

-- private key in OpenSSL PEM format
SshName.PrivateKey = [[-----BEGIN OPENSSH PRIVATE KEY-----
Your Private Key
-----END OPENSSH PRIVATE KEY-----]]

SshName.PrivateKeyPassword = "password" --if needed

--setup timer
function TimerClick()
    --do stuff with ssh:Write as needed
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick

--individual SSH handler functions
SshName.Connected = function()
    print("ssh connected")
    MyTimer:Start(10)
end

SshName.Reconnect = function()
    print("ssh reconnecting...")

end
```



```lua
SshName.Closed = function()
    print("ssh closed")
end

SshName.Error = function(s, err)
    print(string.format("Error: '%s'¥r", error))
end

SshName.Timeout = function()
    print("ssh timeout")
end

SshName.LoginFailed = function()
    print("login failed with " .. string.format("error: '%s'¥r",
error))
end

SshName.Data = function()
    --Handle the data line by line
    line = SshName:ReadLine(Ssh.EOL.Any)
    while line do
        print(line)
        line = SshName:ReadLine(Ssh.EOL.Any)
    end
end

SshName:Connect(address, port, user, password)
```



System

Overview

System は、システム、バージョン、デバッグ情報、およびデバッグ制御に関する情報を提供するグローバルテーブルです。

System.

	Type	Comment
System.BuildVersion	String	X.X.X.X 形式の Composer バージョン
System.MajorVersion	Integer	Composer のメジャーバージョン
System.MinorVersion	Integer	Composer のマイナーバージョン
System.PatchVersion	Integer	Composer パッチのリリースバージョン
System.Build	Integer	Composer のビルドバージョン
System.IsEmulating	Boolean	Composer がオフラインのときにスクリプトが実行されている場合は True
System.IsDebugging	Boolean	デバッグの状態を表すブール値
System.ClearDebugging()	Function	デバッグキャプチャ情報をクリア
System.GetTime()	Function	1970 年 1 月 1 日からの時間をミリ秒単位で返します
System.GetExecutionTime()	Function	現在の 4Hz パスで処理が開始されてから経過したミリ秒数を返します

・System.BuildVersion

「X.X.X.X」形式の Composer バージョン番号を表す文字列。

これは読み取り専用変数であるため、変更を加えても効果はありません。

```
print(System.BuildVersion)                -->> 8.0.1.7
```

・System.MajorVersion

Composer のメジャーバージョン番号を表す整数。 これは、4 桁のバージョン「X.x.x.x」の最初の桁です。

これは読み取り専用変数であるため、変更を加えても効果はありません。

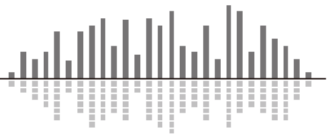
```
print(System.MajorVersion)                -->> 8
```

・System.MinorVersion

Composer のマイナーバージョン番号を表す整数。 これは、4 桁のバージョン「x.X.x.x」の 2 桁目です。

これは読み取り専用変数であるため、変更を加えても効果はありません。

```
print(System.MinorVersion)                -->> 0
```



・System.PatchVersion

Composer パッチのバージョン番号を表す整数。 これは、4 桁のバージョン「x.x.X.x」の 3 桁目です。

これは読み取り専用変数であるため、変更を加えても効果はありません。

```
print(System.PatchVersion)                -->> 1
```

・System.Build

Composer パッチのバージョン番号を表す整数。 これは、4 桁のバージョン「x.x.x.X」の 4 桁目です。

これは読み取り専用変数であるため、変更を加えても効果はありません。

```
print(System.Build)                       -->> 7
```

・System.IsEmulating

オフライン操作の状態を表すブール値。 Composer がオフラインのときにスクリプトが実行されている場合は「true」になり、Composer がオンラインのときにスクリプトが実行されている場合は「false」になります。

これは読み取り専用変数であるため、変更を加えても効果はありません。

```
print(System.IsEmulating)                 -->> false
```

・System.IsDebugging

デバッグの状態を表すブール値。 オフライン実行中は常に「true」ですが、インテリジェントモジュールの右クリックメニューコマンド「オンラインスクリプトデバッグを有効にする」の設定に応じて、オンライン時には「true」または「false」になる場合があります。

これは読み取り専用変数であるため、変更を加えても効果はありません。

```
print(System.IsDebugging)                 -->> true
```

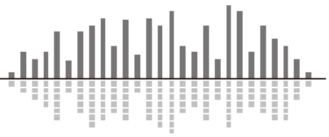
これは、オンラインで「Enable Online Script Debugging」が無効になっている場合に、デバッグ出力ファイルに書き込むデバッグコードの実行を、条件付きで停止するために使用できます。

・System.ClearDebugging()

デバッグ出力ファイルをクリアする機能。

```
System.ClearDebugging()       -- Debug Output File will be cleared
```

これは、長いデバッグセッション中に役立ちます。 一定の時間が経過した後、マイルストーンに達した後、または画面上のボタンをクリックした後に、デバッグ出力ファイルをクリアすることをお勧めします。



・System.GetTime()

1970 年 1 月 1 日から、またはオフラインの場合 PC が起動されてからの時間をミリ秒単位で返します。

```
print(System.GetTime())              -->> -903260332
```

戻り値は、実際には 2 つの 32 ビットの符号なし整数で、合計 64 ビットの符号なし整数を作成します。

通常必要なのは、1 回の戻り値である最下位の数値のみですが、必要に応じて両方にアクセスできます。

```
lowPortion, highPortion = System.GetTime()
print(lowPortion) = 371
print(highPortion) = -903095174
```

32 ビット整数の半値を超えているため、高い部分はマイナス値です。 返される値は通常、それ自体では特に有用ではないため、これは問題ではありません。 ただし、以前の値と比較すると、2 つの時間の間にどれだけの時間が経過したかを判断するために使用できます。 以下の例を参照してください。

System.GetTime()は、オフラインでもオンラインでも同じように計算されますが、Windows コンピュータと Symetrix ハードウェアのタイムゾーン実装の違いによりシフトが見られる場合があります。

表示する実際の日付と時刻を取得する場合は、柔軟性があるため、この目的には os.clock()の方が適しています。

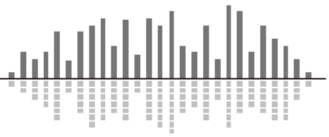
・System.GetExecutionTime()

この API は、スクリプトが消費している CPU 処理時間について詳しく知ることができます。 これは、複雑で集中的なスクリプトを最適化するために必要な場合があり、エラーがスクリプトの CPU 時間を消費しすぎていることを示している場合にのみ必要です。

現在の 4Hz パスで処理を開始してから経過したミリ秒数を返します。 これには、すべてのタイマー、および現在のパスで発生したその他のコールバックが含まれます。 これは、他のルーチンが不足するのを防ぐために、各パスで処理が 1/8 秒(125 ミリ秒)を超えないように制御する必要がある重要なことです。

```
printSystem.GetExecutionTime()          -->> 1
```

スキップされたスクリプトパスまたは負荷率が 100%を超えている場合は、これを使用してスクリプトのプロファイルを作成することをお勧めします。 これをコードのさまざまなセクション(通常はタイマーコールバック)に挿入して、実行に過度に時間がかかっている部分を特定できます。 この情報を使用して最適化できることを願っています。



Usage Examples

次の例は、これらの使用方法を示しています:

・Example 1 –Composer のバージョンを取得する

特定のバージョンの Composer 以降でのみ使用できる機能がいくつかある場合があります。

次のコードは、操作を行う前にこれを確認する方法を示しています。

```
if (System.MajorVersion >= 8) then
   -- Do Something
else
   -- Do Something Else
End
```

・Example 2 – オフライン動作かどうかチェックする

オフラインとオンラインで異なる動作をする必要がある場合があります。 これにより、スクリプトがオフラインで実行されているかオンラインで実行されているかに関係なく、個別のコードパスを作成できます。

たとえば、一般的なアプリケーションの 1 つは、Symetrix ハードウェアでオンラインを実行しているときにプログラムでデバッグをオフにすることです。

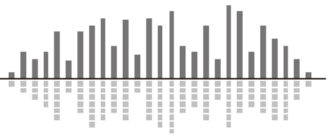
```
if (System.IsEmulating) then
   System.EnableDebugging(true)
else
   System.EnableDebugging(false)
end
```

・Example 3 –ControlIn/Out をオフラインで操作する

オフラインでエミュレートする場合、制御ピンからの入力は使用できません。したがって、System.IsEmulating を使用して、制御ピンからのデータをシミュレートすることもできます。

```
offline = System.IsEmulating
data = 0

if (offline) then
      data = 0.5
else
      data = Controls.Inputs[1].Value
end
--Use the "data"
```

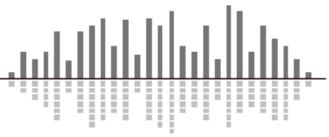
このコードブロックが繰り返し呼び出される場合は、スクリプトの実行中にステータスが変更されないため、System.IsEmulating を変数に設定することをお勧めします。



・Example 4 – 実行されてからの時間経過を取得する

タイマーを使用すると、System.GetTime()を使用して、プロセスを開始してからの経過時間を決定できます。

```
startTime = System.GetTime()

function TimerClick ()
elapsedTime = System.GetTime() - startTime)
--Do something with elapsed time
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(1)
```



TcpSocket

Overview

TcpSocket API を使用すると、TCP を介して他のデバイスと通信できます。

TcpSocket API は、制御ネットワークを使用してのみ機能します。

TCPSocket.

	Return Type	Comment
TcpSocket.New()	TCPSocket object	新しい TCPSocket オブジェクトを作成します

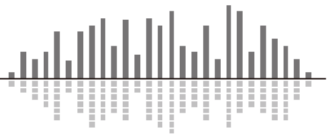
・TcpSocket.New()

TcpSocket.New()は、新しい TcpSocket オブジェクトを作成します。

```
MyTcp = TcpSocket.New()
```

最大 8 つの TcpSocket オブジェクトを、それぞれ同時に使用できます。

ソケットが閉じられると、別の新しい呼び出しで再び使用できるようになります。



TcpSocketName

次のメソッドとプロパティは、「TcpSocketName」という名前の TcpSocket オブジェクトで使用できます。

	Return Type	Comment
TcpSocketName.ID	Integer	スクリプト内のソケット番号（1～8）
TcpSocketName.EventHandler(TcpSocket, event, error)	Callback	ソケットにて、各種イベントが発生したときに呼び出されます
TcpSocketName.ReconnectTimeout	Number	ソケットが外部から切断された場合に、再接続するまで待機する時間（秒）
TcpSocketName.IsConnected	Boolean	ソケットが現在接続中かどうか
TcpSocketName.BufferLength	Integer	受信して読み取られるのを待機しているバイト数
TcpSocketName:Connect(ip, port)	Method	ソケットを、指定 IP アドレスに接続
TcpSocketName:Disconnect()	Method	接続されたソケットを切断
TcpSocketName:Write(data)	Method	接続されたソケットにデータ書き込み
TcpSocketName:Read(length)	Method	接続されたソケットからデータ読み込み
TcpSocketName:ReadLine(EOL, [delimiter])	Method	指定された EOL が検出されるまで、データ読み込み カスタム区切り文字の場合は、文字列として指定
TcpSocketName:Search(pattern, [start])	Method	ソケットの未読データを、開始インデックスから始まる文字列で検索。 戻りはインデックス値を返しますが、データは返しません。
TcpSocketName.Connected(TcpSocket)	Callback	ソケットが接続するときに呼び出されます
TcpSocketName.Reconnect(TcpSocket)	Callback	ソケットが再接続しようとしたときに呼び出されます。
TcpSocketName.Data(TcpSocket, data)	Callback	ソケットに未読データがある場合に呼び出されます。
TcpSocketName.Closed(TcpSocket)	Callback	ソケットが閉じられたときに呼び出されます。
TcpSocketName.Error(TcpSocket, error)	Callback	ソケットにエラーがある場合に呼び出されます。
TcpSocketName.Timeout(TcpSocket, error)	Callback	ソケットにて読み取りまたは書き込みタイムアウトが発生したときに呼び出されます。

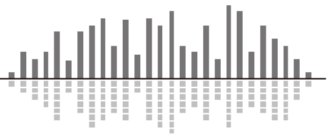
・TcpSocketName.ID

スクリプト内のソケット（1～8）の番号。

```
print(MyTCPSocket.ID)                    -->> 1
```

最大 8 つの TcpSocket オブジェクトを、それぞれ同時に使用できます。

ソケットが閉じられると、別の新しい呼び出しで再び使用できるようになります。



・TcpSocketName.EventHandler(TcpSocket, event, error)

ソケットにて、各種イベントが発生するたびに呼び出される TcpSocket のコールバック関数を割り当てます。

これは 2 つの方法で行うことができます。

```
function TcpHandler (sock, evt, err)
   --handle the event
end

sock.EventHandler = TcpHandler
```

または、匿名関数を使用することもできます

```
sock.EventHandler = function(sock, evt, err)
   --handle the event
end
```

通常、if-elseif-else ステートメントを使用して複数のイベントタイプを処理します。

イベントタイプ「TcpSocket.Events」は、次の6種類です。

Enumeration	Value	Functionality
“Connected”	1	TcpSocket が接続されました
“Reconnect”	2	TcpSocket は再接続を試みています
“Data”	3	TcpSocket には利用可能なデータがあります
“Closed”	4	TcpSocket が閉じました
“Error”	5	TcpSocket でエラーが発生
“Timeout”	6	TcpSocket がタイムアウト

これがどのように使用されるかについては、Example1 を参照してください。

・TcpSocketName.ReconnectTimeout

TCP ソケットの ReconnectTimeout を設定し、デフォルト値の 5 秒を上書きします。 これにより、ソケットが外部で切断された場合に再接続を試行するまで待機する時間が秒単位で構成されます。 待機しない場合は 0 に設定します。

```
mySock.ReconnectTimeout = 1
```

通常はデフォルトで問題ありませんが、接続先のデバイスが再接続するまでに長い時間がかかることがわかっている場合は、これを調整する必要があります。

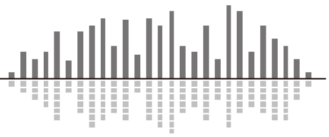
さらに、接続ステータスを手動で監視し、コードに再接続パスを含めることをお勧めします。

・TcpSocketName.IsConnected

ソケットが現在接続されているかどうかを示します。

```
print(mySock.IsConnected)          -->> True
```

これは、接続状態をテストし、適切なアクションを実行するために使用されます。



・TcpSocketName.BufferLength

受信して読み取られるのを待機しているバイト数。

```
print(mySock.BufferLength)          -->> 8
```

・TcpSocketName:Connect(ip, port)

ソケットを IP アドレスとポート(0-65535)に接続します。

これは IP アドレスである必要があります。 ホスト名はサポートされていません。

```
sock:Connect("169.254.179.214", port)
```

デバイステーブルの情報を使用して、正しい IP アドレスが使用されていることを確認できます。

```
currentDeviceIP = Device.RemoteUnit.DanteIP
sock:Connect(currentDeviceIP, port)
```

・TcpSocketName:Disconnect()

接続されたソケットを切断します。

```
sock:Disconnect()                   -- socket is disconnected
```

・TcpSocketName:Write(data)

接続されたソケットにデータのテーブルを書き込みます。

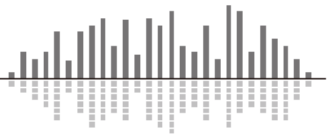
```
sock:Write('V¥x0d')                 -- Data is written and sent
```

・TcpSocketName:Read(length)

接続されたソケットバッファから、指定バイト数のデータを読み込みます。

```
rx = sock:Read(10000)
```

読み取られたバイトはバッファから削除されます。



・TcpSocketName:ReadLine(EOL, [delimiter])

指定された EOL ケース(1～6)が検出されるまで、データを読み取ります。

```
rxLine = sock:ReadLine(1)     --Read until any combination of LF and CR are reached
```

「TcpSocket.EOL」の EOL ケースは、次の通りです。

Enumeration	Value	Functionality
Any	1	CR と LF の任意の組み合わせを検索
CrLf	2	CR または LF を検索
CrLfStrict	3	CR+LF を検索
Lf	4	LF を検索
Null	5	Null(0x00)を検索
Custom	6	カスタム文字列を検索

```
rxLine = sock:ReadLine(TcpSocket.EOL.Any)
--Read until any combination of LF and CR are reached
```

もし「6:Custom」を指定した場合は、オプション 2 番目の区切り文字引数を、文字列として指定します。

```
rxLine = sock:Readline(6, “TheEnd”)
--Read until “TheEnd” is reached
```

・TcpSocketName:Search(pattern, [start])

オプションの 1 ベースの開始インデックスで始まる文字列を、ソケット内の未読データで検索します。

1 ベースのインデックスを返しますが、データは返しません。

```
TcpSocketName:Search(“ImportantMessage”, 1)                -->> 16
TcpSocketName:Search(“AnotherMessage”, 17)                 -->> nil
```

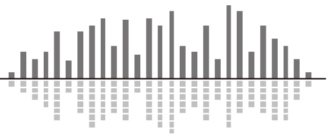
検索では、渡された正確な文字列が検索されます。検索文字列内の特定の文字(*や？など)の解釈はなく、Lua パターンマッチングはサポートされていません。

検索を使用すると、特定の応答を探している状況で効率を向上させることができます。

これは、大量のデータを出力し、1 つのことだけを気にするプロトコルに役立ちます。

各行を読み取って特定の文字列を探すためにデータを解析する代わりに、スクリプトはその文字列を検索できます。

文字列が存在しない場合は、1 つの Read()コマンドで受信したすべてのデータを破棄します。



・TcpSocketName.Connected(TcpSocket)

ソケットが接続するときに呼び出されるコールバック関数を定義します。

これは 2 つの方法で行うことができます。:

```
function ConnectedHandler (TcpSocket)
   --handle the new Connection
end

sock.Connected = ConnectedHandler
```

または、匿名関数を使用することもできます:

```
sock.Connected = function(TcpSocket)
   --handle the new Connection
end
```

これは、一般的な EventHandler コールバックの代わりに使用できます。 後述の Example2 を参照してください。

・TcpSocketName.Reconnect(TcpSocket)

ソケットが TcpSocket テーブルの引数を使用して再接続しようとしたときに呼び出されるコールバック関数を定義します。

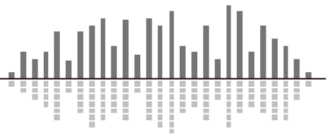
これは 2 つの方法で行うことができます。

```
function ReconnectHandler (TcpSocket)
   --handle the Reconnection attempt
end

sock.Reconnect = ReconnectHandler
```

または、匿名関数を使用することもできます:

```
sock.Reconnect = function(TcpSocket)
   --handle the Reconnection attempt
end
```

これは、一般的な EventHandler コールバックの代わりに使用できます。 後述の Example2 を参照してください。



・TcpSocketName.Data(TcpSocket, data)

TcpSocket テーブルとデータテーブルの引数を使用して、未読データが使用可能な場合に呼び出されるコールバック関数を定義します。

これは 2 つの方法で行うことができます。

```
function DataHandler (TcpSocket, data)
   --handle the data
end

sock.Data = DataHandler
```

または、匿名関数を使用することもできます

```
sock.Data = function(TcpSocket, data)
   --handle the data
end
```

「データ」は、Read 関数または ReadLine 関数を使用して読み取ることができます。

これは、一般的な EventHandler コールバックの代わりに使用できます。 後述の Example2 を参照してください。

・TcpSocketName.Closed(TcpSocket)

TcpSocket テーブルの引数を使用してソケットが閉じられたときに呼び出されるコールバック関数を定義します。

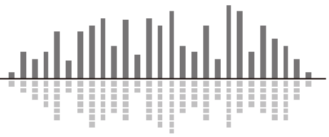
これは 2 つの方法で行うことができます。

```
function ClosedHandler (TcpSocket)
   --handle the socket closing
end

sock.Closed = ClosedHandler
```

または、匿名関数を使用することもできます

```
sock.Closed = function(TcpSocket)
   --handle the socket closing
end
```

これは、一般的な EventHandler コールバックの代わりに使用できます。 後述の Example2 を参照してください。



・TcpSocketName.Error(TcpSocket, error)

TcpScocket テーブルの引数とエラー文字列でエラーが発生したときに呼び出されるコールバック関数を定義します。これは 2 つの方法で行うことができます。:

```
function ErrorHandler (TcpSocket, error)
   --handle the error
end

sock.Error = ErrorHandler
```

または、匿名関数を使用することもできます

```
sock.Error = function(TcpSocket, error)
   --handle the error
end
```

これは、一般的な EventHandler コールバックの代わりに使用できます。 後述の Example2 を参照してください。

・TcpSocketName.Timeout(TcpSocket, error)

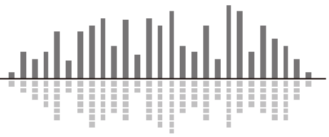
TcpScocket テーブルの引数とエラー文字列を使用して、読み取りまたは書き込みのタイムアウトが発生したときに呼び出されるコールバック関数を定義します。これは 2 つの方法で行うことができます。

```
function TimeoutHandler (TcpSocket, error)
  --handle the Timeout
end

sock.Timeout = TimeoutHandler
```

または、匿名関数を使用することもできます

```
sock.Timeout = function(TcpSocket, error)
   --handle the Timeout
end
```

これは、一般的な EventHandler コールバックの代わりに使用できます。 後述の Example2 を参照してください。



Usage Examples

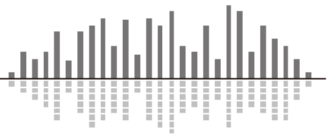
・Example 1 – TCP 接続

この例は、TcpSocket が通常どのように使用されるかを示しています

```
sock = TcpSocket.New()
sock.ReadTimeout = 0
sock.WriteTimeout = 0
sock.ReconnectTimeout = 0
sock.EventHandler = function(sock, evt, err)
      if     evt == TcpSocket.Events.Connected then
            --handle the new Connection
            print("socket connected¥r")
      elseif evt == TcpSocket.Events.Reconnect then
            --handle the Reconnection attempt
            print("socket reconnecting...¥r")
      elseif evt == TcpSocket.Events.Data then
            --handle the data
            rxLine = sock:Read(10000)
            if (nil ~= rxLine) then
                  print("Got:¥r" .. rxLine)
            end
      elseif evt == TcpSocket.Events.Closed then
            --handle the socket closing
            print("socket closed by remote¥r")
      elseif evt == TcpSocket.Events.Error then
            --handle the error
            print(string.format("Error: '%s'¥r", err))
      elseif evt == TcpSocket.Events.Timeout then
            --handle the Timeout
            print("socket closed due to timeout¥r")
      else
            print("unknown socket event¥r")
      end
end

sock:Connect(remoteControlIp, remoteControlPort)
```

上記のイベント列挙テーブルは、受信した「evt」を比較し、適切なアクションを実行するために使用されます。

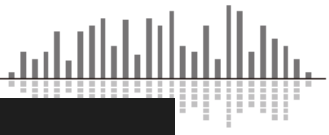
通常、示されているように、6 つのイベントタイプすべてを常に処理する必要があります。



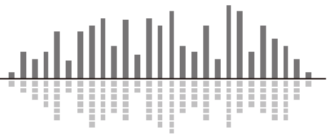
・Example 2 – 個別のハンドラーで TCP を使用する

Example1 で示したように、単一のハンドラーですべてのイベントを処理する代わりに、イベントタイプごとに個別のハンドラーを使用することもできます。if-ifelse の大きなセクションが必要ないため、読みやすくなる傾向があります。

```
sock = TcpSocket.New()
sock.ReadTimeout = 0
sock.WriteTimeout = 0
sock.ReconnectTimeout = 0

sock.Connected = function(TcpSocket)
--handle the new Connection
print("socket connected¥r")
end

sock.Reconnect = function(TcpSocket)
--handle the Reconnection attempt
print("socket reconnecting...¥r")
end

sock.Data = function(TcpSocket, data)
--handle the data
rxLine = sock:Read(10000)
      if (nil ~= rxLine) then
            print("Got:¥r" .. rxLine)
      end
end

sock.Closed = function(TcpSocket)
--handle the socket closing
print("socket closed by remote¥r")
end

sock.Error = function(TcpSocket, error)
--handle the error
print(string.format("Error: '%s'¥r", error))
end

sock.Timeout = function(TcpSocket, error)
--handle the Timeout
```



```
print("socket closed due to timeout¥r")
end

sock:Connect(remoteControlIp, remoteControlPort)
```



Timer

Overview

タイマーAPI は、繰り返しタイマーを設定するために使用されます。 これは、ポーリングを使用して UI を更新し、繰り返しアクションを実行する多くのタイプのインテリジェントモジュールを作成するための基本です。 タイマーが完了するたびに、EventHandler 関数が呼び出されます。

これは通常、UI の状況をポーリングおよび更新したり、定期的にいくつかの繰り返し操作を実行したりするために使用されます。

Timer.

	Type	Comment
Timer.New()		新しいタイマーオブジェクトを作成します

・Timer.New

Timer.New()を使用して、新しい名前のタイマーオブジェクトを作成します。

```
MyTimer = Timer.New()          --Creates a new Timer called MyTimer
```

次に、そのオブジェクトを使用してタイマーを実行します。

インテリジェントモジュールごとに最大 8 つのタイマーを作成できます。

TimerName

次のメソッドとプロパティは、TimerName という名前の Timer オブジェクトで使用できます。

	Type	Comment
TimerName.ID	Int	タイマーオブジェクト番号(1-8)
TimerName:Start(period)	Method	指定された期間(秒)で、タイマーを開始します
TimerName:Stop()	Method	指定されたタイマーを停止します
TimerName.EventHandler(TimerName)	Callback	タイマーが完了したときに呼び出す関数

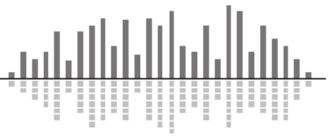
・TimerName.ID

タイマーの ID インデックスを取得します。

```
MyTimer = Timer.New()
MyTimerID = MyTimer.ID                    -- MyTimerID = 1
```

ID は作成順に 1 から割り当てられます。

タイマーの ID は、以下で説明する EventHandler コールバック関数で使用します。



・TimerName:Start(period)

秒単位で指定された期間で、タイマーを開始します。

```
MyTimer:Start(1)        --Starts the timer with a period of 1 second
```

タイマーは最大 4Hz で更新されるため、最小設定時間は 0.25 秒です。

・TimerName:Stop

指定されたタイマーを停止します。 Start メソッドで再始動できます。

```
MyTimer:Stop            --Stops the timer
```

・TimerName.EventHandler(TimerName)

タイマーが設定時間に達するたびに呼び出される関数を割り当てます。

これは 2 つの方法で行うことができます。

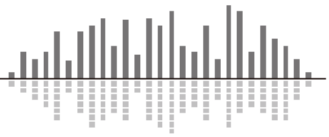
```
function TimerClick ()
   --Do lots of stuff every time
end

MyTimer.EventHandler = TimerClick
```

または、匿名関数を使用することもできます。

```
MyTimer.EventHandler = function ()
   --Do lots of stuff every time
end
```

オプションで、Timer オブジェクトを関数の引数として渡すことができます。 これにより、Timer.ID にアクセスできるようになり、複数のタイマーに同じコールバック関数を使用できるようになります。

```
function TimerClick (TimerName)
   if TimerName.ID == MyTimerID then
         --Do stuff for the first timer
   else
         --Do different stuff for every other timer
   end
end

MyTimer.EventHandler = TimerClick
MyTimerID = MyTimer.ID

MyOtherTimer.EventHandler = TimerClick
```



Usage Examples

次の例は、これらの使用方法を示しています。

・Example 1 – タイマーを使用する

この例は、タイマーを使用して UI を更新したり、定期的に繰り返しアクションを実行したりする方法を示しています。

```
function TimerClick ()
   --Do lots of stuff every time
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(.25)
```

まず、タイマーが完了するたびに、呼び出す関数を定義します。

次にタイマーを作成し、関数をタイマーの EventHandler として設定します。 最後にタイマーを開始します。

・Example 2 – フェーダーの値を取得して LED を光らせる

TimerClick()関数内に配置する、最も一般的な処理の 1 つは、各コンポーネントの入力値を取得し、それらを使用して何かを実行することです。

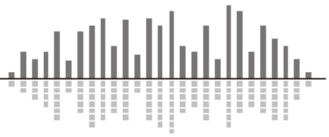
```
function TimerClick ()
   currentFaderValue = NamedControl.getValue("MyFader")
   if currentFaderValue > .5 then
         NamedControl.setValue("MyLED", 1)
   end
end
```

通常、このタイプのタイマーを停止することはありません。 ただし多くの場合、各 UI 要素が実際に変更されているかどうかを確認し、変更されている場合にのみ処理を実行する必要があります。

・Example 3 – タイマーのイベントを一度だけ実行する

タイマーイベントのアクションを 1 回だけ実行する場合は、TimerClick()関数の最後に Timer.Stop()を追加します。

```
function TimerClick ()
   --Do lots of stuff one time after 10 seconds
MyTimer:Stop() --Stop the Timer.
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(10)
```



・Example 4 – 複数のタイマーを使用する

異なる間隔で複数のタイマーが必要な場合は、複数のタイマーを作成するだけです。

```
function FastTimerClick ()
   --Do stuff that needs to be done very often
end

function SlowTimerClick ()
   --Do other stuff that can happen less regularly
end

FastTimer = Timer.New()
FastTimer.EventHandler = FastTimerClick
FastTimer:Start(.25)                -- Calls EventHandler every .25s

SlowtTimer = Timer.New()
SlowTimer.EventHandler = SlowTimerClick
SlowTimer:Start(10)                 -- Calls EventHandler every 10s
```

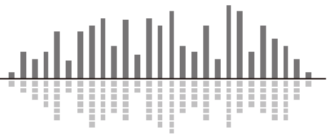
これは、単一のイベントハンドラーで交互に処理できます。

```
function TimerClick (ID)
   if ID == 1 then
         --Do stuff for the fast timer
   elseif ID == 2 then
         --Do different stuff for the slow timer
   end
end

FastTimer = Timer.New()
FastTimer.EventHandler = TimerClick
FastTimer:Start(.25)                -- Calls EventHandler ever .25s

SlowtTimer = Timer.New()
SlowTimer.EventHandler = TimerClick
SlowTimer:Start(10)                 -- Calls EventHandler ever 10s
```

これは、UI のポーリングと更新を低速で行いながら、一部の内部制御処理を高速で更新する場合に役立ちます。

しかし、これはほとんどの場合、このニーズを処理するための最も効率的な方法ではありません。

代わりに、次の例を参照してください。

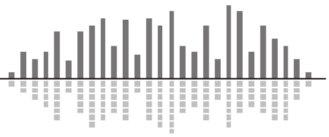


・Example 5 – 複数のタイマーを使用する2

上記の例で説明した、間隔の違うタイマーを複数使用する方法より効率的な方法は、1つの高速タイマー関数の中にループカウンターを配置することです。

```
timerMultiplier = 10
timerCounter = 0

function TimerClick ()
      -- Do stuff every timer the timer occurs

      timerCounter = timerCounter + 1

      if timerCounter >= timerMultiplier then
            --Do other stuff that can happen less regularly
            timerCounter = 0              --Reset the timerCounter
      end
end

MyTimer = Timer.New()
MyTimer.EventHandler = TimerClick
MyTimer:Start(.25)
```

これは、より少ない処理で効率的になりますが、
低速タイマーが高速タイマーの倍数である場合に限り使用できます。



UdpSocket

Overview

UdpSocket は、UDP を介した他のデバイスとの通信を可能にするグローバルテーブルです。

UdpSocket API は、コントロールまたはオーディオネットワークを使用して機能します。

この API は通常、デバイスのカスタム UDP 制御プロトコルを使用してデバイスと通信するために使用されます。

通常、データをソケットとの間で送受信する前に、UdpSocket を作成して開く必要があります。

UdpSocket.

	Return Type	Comment
UdpSocket.New()	UdpSocket object	新しい UdpSocket オブジェクトを作成します

・UdpSocket.New()

UdpSocket.New()は、新しい UdpSocket オブジェクトを作成します。

```
MyUdp = UdpSocket.New()
```

最大 8 つの UdpSocket オブジェクトを、それぞれ同時に使用できます。

ソケットが閉じられると、別の新しい呼び出しで再び使用できるようになります。

UdpSocketName.

次のメソッドとプロパティは、UdpSocketName という名前の UdpSocket オブジェクトで使用できます。

	Return Type	Comment
UdpSocketName.ID	Integer	スクリプト内のソケット番号(1～8)
UdpSocketName:Open(ip, port)	Method	ソケットを指定 IP アドレスにバインドします
UdpSocketName:Close()	Method	ソケットを閉じます
UdpSocketName:Send(ip, port, data)	Method	データを特定の IP およびポートに送信します
UdpSocketName:GetSockName()	Method	ソケットの現在の IP アドレスとポートを返します
UdpSocketName.Data(ip, port, data)	Callback	データを受信したときに呼び出される関数

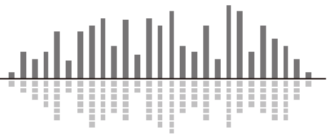
・UdpSocketName.ID

スクリプト内のソケット(1～8)の番号。

```
print(MyUdp.ID)                    -->> 1
```

最大 8 つの UdpSocket オブジェクトを、それぞれ同時に使用できます。

ソケットが閉じられると、別の新しい呼び出しで再び使用できるようになります。



・UdpSocketName:Open(ip, port)

ソケットを IP アドレスとポート(0-65535)にバインドします。

```
MyUdp:Open("192.168.1.20", 48631)
```

オンラインで使用する場合、スクリプトで制御ネットワーク IP アドレスまたはオーディオ IP アドレス以外のものを指定すると、ポートが正しく開かれません。 スクリプトは、デバイス API によって報告されるオーディオまたは制御ネットワークの IP を指定する必要があります。

```
MyUdp:Open(Device.LocalUnit.ControlIP, 48631)
MyOtherUdp:Open(Device.LocalUnit.AudioIP, 48631)
```

または、スクリプトで IP アドレスに「0.0.0.0」が指定されている場合は、制御ネットワークが自動的に使用されます。

```
MyUdp:Open("0.0.0.0", 48631)
```

この自動オプションにより、電源投入時に IP アドレスが取得される前にバインドできます。 それ以外の場合、スクリプトは、デバイステーブルに有効な IP アドレスが存在するまで待機してから、番号付きポートを開く必要があります。

特定のポート番号が必要ない場合は、ポート 0 を指定することにより、デバイスによってポートが自動的に選択される場合があります。

```
MyUdp:Open("192.168.1.20", 0)
```

一部のデバイスはそれらを呼び出したポートに応答しますが、他のデバイスは特定のポートにのみ送信することに注意してください。 これにより、一部のデバイスで自動ポート選択を使用できなくなる場合があります。

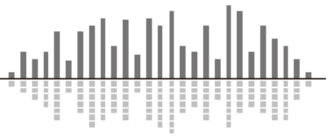
複数のソケットが必要な場合、たとえば、応答を別々に保つ 2 つの異なるデバイスを制御する場合は、2 つの異なるポートが使用されていることを確認するか、デバイスにポートを自動的に割り当てさせます。

```
--Bad
MyUdp:Open(Device.LocalUnit.ControlIP, 48631)
MyOtherUdp:Open(Device.LocalUnit.ControlIP, 48631)

--Good
MyUdp:Open(Device.LocalUnit.ControlIP, 48631)
MyOtherUdp:Open(Device.LocalUnit.ControlIP, 4001)

--Good
MyUdp:Open(Device.LocalUnit.ControlIP, 0)
MyOtherUdp:Open(Device.LocalUnit.ControlIP, 0)
```

最後に、このメソッドは、指定された IP とポート番号でポートを作成できる場合、「true」を返します。 これには、有効な IP、ポート番号、そのポートと IP でまだ作成されていないもの、および UDP ポートが多すぎないことが必要です。

これは、オープニングが成功したことをテストするために使用できます。

```
udpOpen = StationUdp:Open(Device.LocalUnit.AudioIP, 0)
if udpOpen then
--do some stuff
end
```



・UdpSocketName:Close()

ソケットを閉じて、別の割り当てに使用できるようにします。 既存のオブジェクトを再利用してはなりません。

```
MyUdp = UdpSocket.New()
--do a bunch of stuff

MyUdp:Close()
MyNewUdp = UdpSocket.New()
```

・UdpSocketName:Send(ip, port, data)

特定の IP およびポート(0-65535)に、指定されたデータを送信します(パケットに収まる必要があります)。

```
MyUdp:Send("192.168.1.101", 48631, "CS 1 0¥x0d¥x0a")
```

デバイステーブルの情報を使用して、正しい IP アドレスが使用されていることを確認できます。

```
MyUdp:Send(Device.RemoteUnit.DanteIP, 48631, "CS 1 0¥x0d¥x0a")
```

上記のように 0.0.0.0 または Device.LocalUnit.ControlIP を使用して IP ソケットを開くことができますが、データを送信する前に、Device.LocalUnit.ControlIP がゼロ以外であるかどうかを確認する必要があります。

次の GetSocketName()を参照してください。

・UdpSocketName:GetSockName()

ソケットの現在の IP アドレスとポートを返します。

これは、ソケットが自動ポートにバインドされている場合に使用されているポート情報を取得するのに役立ちます。

有効な IP が割り当てられるまで、送信は機能しないため、これをチェックとして使用できます。

```
receivedIP, receivedPort = MyUdp:GetSockName()
print("IP: " .. receivedIP .. " Port: " .. receivedPort)
```

・UdpSocketName.Data(socket, packet)

データを受信したときに呼び出されるコールバック関数。

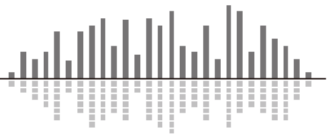
ソケットオブジェクトとデータパケットテーブルの 2 つの引数を受け入れる必要があります。

```
function HandleData(socket, packet)
   -- Do Stuff
end

MyUdp.Data = HandleData
```

ソケットオブジェクトは、メッセージが受信されたソケットに関する情報を提供します。

データパケットテーブルには、受信データを構成する 3 つの要素があります。

.Address	受信したパケットの送信元 IP アドレス
.Port	受信したパケットの送信元ポート
.Data	受信したパケットのデータペイロード



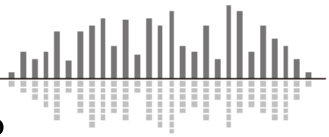
Usage Examples

次の例は、これらの使用方法を示しています。

・Example 1 – UDP ソケット

この例は、UdpSocket が通常どのように使用されるかを示しています。

```
function HandleData(socket, packet)
   --Info about receiving socket and packet
   print("Socket ID: " .. socket.ID)
   receivedIP, receivedPort = socket:GetSockName()
   print("Socket IP: " .. receivedIP)
   print("Socket Port: " .. receivedPort)
   print("Packet IP: " .. packet.Address)
   print("Packet Port: " .. packet.Port)

   --Do stuff with the received packet
   print("Packet Data: ¥r" .. packet.Data)
end

MyUdp = UdpSocket.New()
MyUdp:Open(Device.LocalUnit.ControlIP, 0)
MyUdp.Data = HandleData
MyUdp:Send("192.168.1.101", 48631, "CS 1 0¥x0d¥x0a")
```



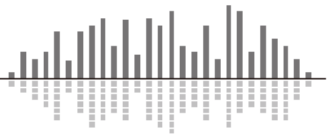
・Example 2 – Device.LocalUnit.IP が割り当てられてからデータを送信する

この例は、Device.LocalUnit.ControlIP が有効であることをテストしてから使用することで改善できます。

これが必要なのは、電源投入時に、構成によっては IP アドレスの割り当てに時間がかかる場合があるためです。

したがって、Timer EventHandler 内でこれらのチェックを実行して、IP アドレスの準備ができたときにのみ、UDP メッセージを繰り返しチェックして送信を開始します。

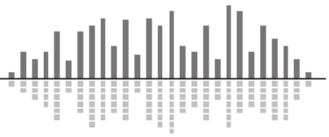
```
udpInitialized = false
MyUdp = UdpSocket.New()

function UDPStartTimerClick ()
    --check if initialized, i.e IP valid and UDP port open
    if udpInitialized == false then
        --before can send message need to determine IP address is valid IP address
        if Device.LocalUnit.ControlIP ~= nil then
            --have an IP Address
            print("Valid IP: " .. tostring(Device.LocalUnit.ControlIP))
            MyUdp:Open(Device.LocalUnit.ControlIP, 0)
            MyUdp.Data = HandleData
            udpInitialized = true
           --have done initial setup, so set this as true so don't need to do again.

         else
            --IP address not ready, Try again next Timer pass
            print("Not ready Online Path with Device IP: "
                   ..tostring(Device.LocalUnit.ControlIP))
        end

    else
        --UDP Socket is open with valid IP,
        --Now can send the data and do all the work each timer pass
        MyUdp:Send(otherUnitIP, port, messageSendFlashUnit)
    end
end

UDPStartTimer = Timer.New()
UDPStartTimer.EventHandler = UDPStartTimerClick
UDPStartTimer:Start(1)
```



・Example 3 – 送信元に応じて異なる操作を実行する

このソケットオブジェクトとデータパケットの.Address 要素と.Port 要素を使用して、データを処理するときにパケットの送信元を判別し、送信元に応じて異なるアクションを実行できます。

```
function HandleData(socket, packet)
   --Do Stuff with the received packet
   if      packet.Address == source1 then
          --Do stuff with the packet data
   elseif packet.Address == source2 then
          --Do different stuff with the packet data
   end
end
```



Lua References and Best Practices

Composer の Intelligent Module は、その機能を制御しデバイスと通信するためにスクリプトを利用します。これらのスクリプトは、強力で効率的、かつ軽量なスクリプト言語である Lua で記述されています。Lua はコンピュータと Symetrix DSP のような組み込みデバイスの両方で動作するソフトウェアと対話するスクリプトを書くことができるため、他の同様のアプリケーションで使用されています。

Lua の全体像を説明することは、Symetrix のドキュメントの範囲を超えています。その代わりに、このドキュメントでは、自由に利用できる参考資料を紹介します。

Refetence Manual

以下の Lua リファレンスマニュアルは Intelligent Module を作成する場合最も重要な資料です。

http://www.lua.org/manual/5.3

このページは Lua 言語とすべての標準ライブラリ関数に関する完全なリファレンスです。

また Composer 8.x は Lua 5.3 を使用していますが、将来的には v5.4 以降へアップデートするかもしれません。

Programming In Lua

Lua をより深く学ぶためには、公式の「**Programming In Lua**」という本を強くお勧めします。

この本は、紙と電子書籍の両方の形式で提供されています。

http://www.lua.org/pil/

Composer Removed Features

Composer では、ユーザーとシステムを保護するために、下記の Lua 標準ライブラリ機能が無効になっています。

- **LuaSockets**

  LuaSockets は、ソケットを実装したパブリックな Lua ライブラリです。DSP ユニットのファイルシステムで利用できるようにすることは、セキュリティ上のリスクがあると考えられているため含まれていません。

  この機能の多くは、TCP および UDP API Extensions で利用可能です。

- **SSL**

  SSL はサポートされていません。

- **IO**

  io ライブラリは、セキュリティと信頼性の観点からサポートされていません。

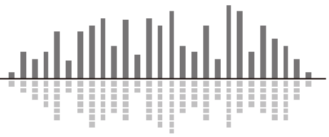
- **OS**

  セキュリティと信頼性の観点から、os ライブラリは一部のみサポートされています。

  "execute", "exit", "remove", "rename", "setlocale", "tmpname" 関数は動作しないようになっています。

- **DEBUG**

  デバッグライブラリは、現在、実行時間の監視を可能にするためにサポートされています。



この製品の取り扱いなどに関するお問い合わせは株式会社オーディオブレインズまでご連絡ください。お問合せ受付時間は、土日祝日、弊社休業日を除く 10:00～18:00 です。

株式会社オーディオブレインズ

〒216-0034
神奈川県川崎市宮前区梶ヶ谷 3-1
電話：044-888-6761
https://audiobrains.com

AUDIO BRAINS