

# FLIGHTDECK TECHNOLOGY

## SIM-board Custom Coding

### User Guide

Draft 1.0 - 20<sup>th</sup> January 2008

Always make backups of any vital files before modifying them!  
There may be errors in this document - check revision history for the latest changes.

# SIM-board Scripting Language

## Basic User Guide

### Introduction

This is a brief guide for advanced users who wish to add their own functionality to their SIM-board project files. Topics covered include: the basics of the SIM-board Scripting Language, project file sections, establishing links to 3rd-party applications, assigning events to input nodes, and controlling the display of output nodes.

This guide assumes good familiarity with the SIM-board Universal Controller software (**version 1.0s BETA or higher**) with existing projects using point-and-click assigned node functions, and also assumes some knowledge of programming, although this is not essential if you are keen to learn!

This guide is written not as a programmer's reference or syntax manual, but more as read-and-do guide with examples and explanations.

### Document History

1.0	20 January 2008	Initial issue
-----	-----------------	---------------

### Topics Covered

- SIM-board Scripting Language: The Basics
- Script Structure
- Declaring Global Variables
- Initializing links to ActiveX controls (including FSUIPC)
- Accessing SIM-board modules
- Assigning events to Switch nodes
- Introducing the “This” variable
- Assigning Events to Rotary Encoder nodes
- Synchronizing Rotary Encoder variables with a 3rd-party application
- Defining the encoder change action
- Calling Procedures and Functions
- Controlling output nodes - basic LED example
- Setting 7-segment display digits
- Advanced segment setting - Direct setting of digit segments
- Reading and Writing Bits instead of Bytes
- Using a switch to modify a bit parameter in PMSystems
- Using Include files
- Using potentiometers
- Controlling a potentiometer configured as a multiple zone “Pot as Switch” type
- Sending Keystrokes
- Importing 3rd-party DLL functions into your project
- Input repeater events
- SIM-board Scripting Language: Useful SIM-board functions
- SIM-board Scripting Language: Built-in functions and procedures
- Quick Reference

## SIM-board Scripting Language: The Basics

The SIM-board Scripting Language (SSL for short) is a VisualBasic-like script engine built in to the SIM-board Universal Controller (SUC for short) software. SUC project files (\*.sbp files) are made up of lines of code in the SSL format.

The SSL code provided in any given project file determine how your SIM-boards and your 3rd-party programs interact. You can control what happens when an input event occurs (for example, a switch changing state, or an encoder being twisted) and you can also control the output state of output nodes (for example, turning an LED node on or off depending on a certain condition, or writing a sequence of numbers to a 7-segment display block).

### Script Structure

The script file for any given project is accessed by clicking on *Project | View Code* from the SUC main menu. There will be at least 4 tabs shown, and if you reference other files (known as *include* files) then they will also be shown in their own tab.

#### **Global Declarations Code**

This section is where you must declare all your global variables, make certain initialization calls, and establish links to 3rd-party ActiveX controls or import any DLL procedures and functions you wish to use from 3rd-party DLLs. This global declarations part of the script is the first part of code to be executed when *Run Project* is chosen. It is executed only once. All projects must specify some code in this section - it is a **COMPULSORY** section of code.

#### **Power Up Code**

This section contains lines of code that must be executed only when all the SIM-boards have been positively identified during the board bootup sequence after *Run Project* is chosen. This section of code is executed only once, and occurs before entering the main loop procedure (see below). All variable declarations made in this part of the code remain local to this code only - they are not available for use in any other code section. Typical uses for this section include initializing script variables to the current values of 3rd-party applications, based on the current states of certain SIM-board nodes. This code section is **OPTIONAL** and does not typically have to be used.

#### **Loop Code**

This section contains lines of code that are executed in a constant loop for as long as the project remains running. The loop is automatically managed and optimized by the SUC. Typical uses for this section include the continuous reading of 3rd-party application variables, and setting of SIM-board output node states. This section is **OPTIONAL** but would typically be used whenever control of output nodes is required.

#### **Finalize Code**

This section is executed when *Stop Project* is chosen, or whenever abnormal termination of the script occurs. It is typically used to close links to 3rd-party ActiveX controls, or perform tidying up functions related to the SIM-boards (for example, to set all LED states to off). This section is **OPTIONAL**. For projects which make use of FSUIPC, we must use this section to close the link to FSUIPC here.

## Declaring Global Variables

Global variables are variables available for use throughout your script (they are distinct from local variables, which as their name suggests, are only available for use locally within a procedure or function). Global variables can only be declared in the *Global Declarations Code* section.

```
Dim myVariable
Dim anotherVariable as Integer
Dim yetAnotherVariable = 0
```

The above shows 3 different ways of declaring a variable. The first is a simple declaration of a variable of no specific type. The second is a variable declared for use as an Integer (whole number). The third is a variable declared with an initial value of zero (and typecast to the default type, in this case, an integer too).

Once declared, variables can be assigned values:

```
Dim myVar = 0
myVar = 123

Dim anotherVar = 456
Dim totalVars
totalVars = myVar + anotherVar
```

Above, 3 variables are declared and assigned values in different ways.

Typical uses for declaring global variables are when such variables are required to hold information that is to be accessible throughout the script, including within custom sub procedures or functions, or from within the *Power Up*, *Loop* or *Finalize* code sections.

Note that variables can be named using any combination of alphanumeric characters and the “\_” character, the only limitation being that the variable name cannot start with a number.

Examples:

```
Dim myVar [valid]
Dim myVar2 [valid]
Dim 3WaySwitch [not valid]
```

Module names (see *Accessing SIM-board modules* below) should also not have a digit character as the first character of the name.

## Initializing links to ActiveX controls (including FSUIPC)

One method for accessing 3rd-party applications from your SSL project is to call methods and functions from a 3rd-party ActiveX control. The SUC software includes an ActiveX DLL specially written to help access to FSUIPC functions from within SSL code. To access the ActiveX DLL, you must declare a global variable in the *Global Declarations* section as follows:

```
Dim fs = new ActiveXObject("SBMSFS.FSUIPC")
```

The variable *fs* is now an object representing the ActiveX control. Before using it, you must call the *Init* method of the control to ensure a link to FSUIPC is established:

```
fs.Init
```

[Note that Flight Simulator with FSUIPC installed, or WideFS, must be running at the time that this code is executed (ie. on clicking *Run Project*) in order for the *Init* call to succeed in establishing a link to FSUIPC. Clearly, if FSUIPC (or WideFS if operating over a network) is not available at the point this code is executed, then the link will not be established and your project will not work.]

From now on, you can access functions from the *fs* object to read and write data to offsets in the FSUIPC mapping space (see later for a specific example).

## Finalizing links to ActiveX controls (including FSUIPC)

When using ActiveX controls, they may need to clean up at the end of their usage within a script project. If this is required, you should call the control's *Finalize* event here, in this *Finalize* section of code. For the supplied FSUIPC ActiveX control, we must do this:

```
fs.Finalize
```

This ensures that the ActiveX control has a chance to close down properly. In this particular case, the *fs.Finalize* method unlinks our project from FSUIPC. If we didn't do this, then subsequent runs of our project would fail to initialize with FSUIPC and you would not get any interaction between your project and FSUIPC, because the link was not closed properly on the previous run. To fix this, you'd have to quit and reload the SIMboards.exe program and run your project again. By calling *fs.Finalize* in the *Finalize* section of code, we can be sure that we keep our link to FSUIPC available during all subsequent runs of our project without having to reload the SUC software. It is also good programming practice to clear up after your project has finished!

## Accessing SIM-board modules

Each module currently listed in the SUC software is available for referencing within code, to access the objects of that module. Each module is assigned a variable name: if you have not yet named your module within the SUC software itself, the module will be automatically declared with a name of *MyBoard* followed by the 12 digit serial number of the module. If you have named your module, it will be declared for use with the name you have chosen, filtered for spaces and other invalid characters.

Example: a module named in the SUC software as *My Overhead Inputs* would be accessible in script using the variable *MyOverheadInputs*.

Note that SIM-board modules are automatically declared for you, but the declarations will not appear in any code section under *Project | View Code*. You do not need to declare the modules yourself.

## Assigning events to Switch nodes

Switch nodes can be configured to call a given procedure or function, or execute a given code statement, when the switch turns on, turns off, or changes state. Assignment of events to execute for each node must be declared in the *Global Declarations* section.

Syntax:

```
<the input module name>.Input[<node number>].OnStateOn = "<statement to execute><;another statement>"
```

where *<the input module name>* is the name you have given to the input module, *<node number>* is the switch node number (index from 1), and *<statement to execute>* is a valid SSL code statement. Optionally, further statements can be called by separating all statements with a semi-colon.

Example: to write a value of 1 to offset 0x0BC8 for 2 bytes (representing the parking brake going on) when the switch goes on, do this:

```
MyInputModule.Input[1].OnStateOn = "fs.BeginRequest;fs.WriteStateData(0x0BC8, 2, 1);fs.SendRequest"
```

When the switch goes on, 3 statements will be executed:

### **fs.BeginRequest**

[This call references the SIM-board FSUIPC ActiveX control and must be called before any *ReadStateData* or *WriteStateData* call is made. The *BeginRequest* can be thought of as initializing a fresh FSUIPC request.]

### **fs.WriteStateData**

[Another FSUIPC function, and as the name suggests, allows writing of data to the simulator. In this case, we are writing "2" bytes (the 2<sup>nd</sup> parameter) to memory location "0x0BC8" as defined in the FSUIPC Programmers Guide (1<sup>st</sup> parameter), with a value of "1" (3<sup>rd</sup> parameter). For this particular FSUIPC offset, a value of 1 means TRUE (ie. set the parking brake), while if you wanted to release the parking brake you would write "0" here instead (meaning FALSE, or off).]

## fs.SendRequest

[This call means we have finished requesting data or sending data to FSUIPC, and that we now want to perform the requested operations. Upon calling this statement, FSUIPC will process the *ReadStateData* and *WriteStateData* items that have been called.]

In this particular case, the switch is only setting one state within the simulator. If we wanted to extend the functionality of this switch, we could do the following:

```
MyInputModule.Input[1].OnStateOn = "fs.BeginRequest;fs.WriteStateData(0x0BC8, 2, 1);fs.SendRequest"  
MyInputModule.Input[1].OnStateOff = "fs.BeginRequest;fs.WriteStateData(0x0BC8, 2, 0);fs.SendRequest"
```

This example adds the *OnStateOff* declaration for the same switch, and this time we are writing a value of 0 to the offset, to turn the parking brake off when the switch goes off.

A further example can also be considered:

```
MyInputModule.Input[1].OnStateChange = "fs.BeginRequest; fs.WriteStateData(0x0BC8, 2, MyInputModule.Input[1].State); fs.SendRequest"
```

Note here that we have defined only 1 event for this switch (*OnStateChange*) and that the value passed to FSUIPC in the *WriteStateData* call directly refers to the state of the switch itself (*MyInputModule.Input[1].State*). The *State* property of the switch holds a value of 1 when the switch is on, and 0 when the switch is off. Hence when the switch changes state, this code is executed and sets the parking brake to the current state of the switch.

## Introducing the “This” variable

A further example of the *OnStateChange* code line can be considered:

```
MyInputModule.Input[1].OnStateChange = "fs.BeginRequest; fs.WriteStateData(0x0BC8, 2, This.Element.State); fs.SendRequest"
```

Note that the value passed to the *WriteStateData* function is *This.Element.State*. Whenever a specific input type is called (in this case a switch, but this also applies to rotary encoder and potentiometer node types too), then *This* refers to the board itself (in this case, *This* points to *MyInputModule*); *Element* refers to the element changing (in this case, *Input[1]*), and *State* refers to the present state of the switch. Using *This.Element.State* allows you to refer to the switch state itself but without hard-coding the name of the module or the node number, thus allowing you to easily change the node number or name of the module in future without having to worry about changing the same within the code statement also.

## Assigning Events to Rotary Encoder nodes

Encoder nodes can be configured to call a given procedure or function, or execute a given code statement, when the encoder changes position. Assignment of events to execute for each node must be declared in the *Global Declarations* section.

Syntax:

```
<the input module name>.Rotary[<node number>].OnChange = "<statement to execute><;another statement>"
```

where *<the input module name>* is the name you have given to the input module, *<node number>* is the encoder node number (index from 1), and *<statement to execute>* is a valid SSL code statement. Optionally, further statements can be called by separating all statements with a semi-colon.

Example: to increase an internal counter when the encoder is changed:

```
Dim myCounter = 0  
  
MyInputModule.Rotary[1].OnChange = "myCounter = myCounter + This.Element.ValueChange;  
debugprint Str(myCounter)"
```

First, an integer type variable is declared (this will hold our internal counter number). The *OnChange* property of the *Rotary[1]* node is then assigned some code to be executed when the encoder changes state:

**myCounter = myCounter + This.Element.ValueChange**

[This simply increments the *myCounter* variable with the value of the change in encoder position. *This.Element* refers to *MyInputModule.Rotary[1]* and *ValueChange* is an integer representing the number of detents that have been detected, signed as appropriate for clockwise (positive value) or anticlockwise (negative) movement. Typically, a single detent change in position of the encoder will place a value of 1 (clockwise) or -1 (anticlockwise) into *ValueChange*. If the Fast Slew and Fast Time parameters have been met (to allow for simulated fast encoder twisting), then *ValueChange* will hold the Fast Slew value (default of 5), signed again for direction.]

**debugprint "Counter is " & Str(myCounter)**

[The *debugprint* procedure allows outputting of a string to the Debug tabbed window towards the bottom of the SUC main window area (visible only when there are 1 or more debug messages available). In this case, we are outputting a simple string of "Counter is " plus the *myCounter* integer variable converted to a string with the *Str()* function.

On running such a project and turning the encoder, the debug window will show the value of the *myCounter* variable.

## Synchronizing Rotary Encoder variables with a 3rd-party application

One of the main reasons for using a rotary encoder within a simulator environment is for changing autopilot variables such as heading, altitude, speed and vertical speed, as well as communication and navigation radio frequencies, GPS controls, and other such items. All these items will have an initial value by the time the project is first run; for example, the autopilot altitude value may currently be set to 5000. If we want to assign an encoder to increment and decrement this value, we must first ensure that the starting value currently set in the simulator

program is obtained and used as the starting value, to allow us to modify that value when the encoder is changed and write back the new value to the simulator. To do this, we must configure the SUC software to obtain this starting value, and to maintain it throughout the running of the project at a chosen interval (this allows use of the mouse or other method of changing the simulator item's value other than by using the encoder, while ensuring that such changes are detected in a timely manner and our project starting value updated).

In the *General Declarations* section:

```
Dim fsvar_07D4
Dim fsvar_apaltitude
```

In the *Loop* section:

```
fs.BeginRequest
fs.ReadStateDataNew(0x07D4, 4, GetPtr(fsvar_07D4, 3))
fs.SendRequest
```

The above declares 2 variables, one to hold the value returned from FSUIPC location 0x07D4 and the other to hold that value converted to feet (refer to FSUIPC Programmer's Guide).

During the loop procedure the *ReadStateDataNew* function is called to obtain the value held in FSUIPC at location 0x07D4 (representing the autopilot altitude value). "0x07D4" (1<sup>st</sup> parameter) is the offset, "4" is the number of bytes to read, and "GetPtr(fsvar\_07D4, 3)" returns a pointer to our previously declared variable *fsvar\_07D4* to allow FSUIPC to store the value returned directly into it.

As this loop is constantly executed during project running, it follows that the value held in *fsvar\_07D4* variable is constantly being updated.

Additionally, in the *General Declarations* section:

```
MyInputModule.Rotary[1].Idle.Setup(2000, TRUE, "fsvar_apaltitude =
Cint((fsvar_07D4/65536)*3.28084)")
```

The rotary encoder node has an *Idle* property, which will execute at a defined time interval, whenever the encoder has been idle (ie. not used) for the specified time period. In this case, the code is executed every 2000 milliseconds (2 seconds). An explanation of the parameters follows:

**2000**

[time interval for the idle code execution]

**TRUE**

[auto enable: this will set up the idle code to be executed only when the rotary encoder "OnChange" property has come code defined]

**fsvar\_apaltitude = Cint((fsvar\_07D4/65536)\*3.28084)**

[this converts the value held in *fsvar\_07D4* into something useful and understandable to us, in this case, a value in feet. The conversion is as specified in the FSUIPC Programmer's Guide, and is surrounded by the *Cint* function which converts the result of the calculation from a floating point number (a number with a fractional part) to an integer number (a number without any fractional part). Hence the variable *fsvar\_apaltitude* will contain the value of the autopilot altitude setting in the simulator.

By setting up this idle state we can be sure that the value held in *fsvar\_apaltitude* will always be

the simulator's autopilot altitude value in feet. If the value is ever changed by mouse or keyboard or any other method not associated with your project, then we can be sure that within 2 seconds our project will always contain the new value as the base reference for our yet-to-be-defined encoder change action.

## Defining the encoder change action

Now that we have a starting value for our encoder value, we can define the action to be taken when the encoder is changed.

In the *General Declarations* section:

```
MyInputModule.Rotary[1].OnChange = "ChangeAutopilotAltitudeValue(fsvar_apaltitude,  
This.Element.ValueChange*100, 0x07D4, 4)"
```

This statement will now be executed each time the encoder is changed. In doing so, the code that is executed is slightly different to the examples given so far in this manual - the call to *ChangeAutopilotAltitudeValue* is a reference to a procedure which will contain all the code necessary for working out the new value in feet that we want to set, and to convert that value to the correct value to write back to FSUIPC in FS units. We will now learn how to define our own procedures...

## Calling Procedures and Functions

You can create custom procedures and functions to be called from anywhere in your script to do whatever you wish. In the above example, we will define a procedure called *ChangeAutopilotAltitudeValue* that will take various parameters and write a new value back to the FSUIPC.

In the *General Declarations* section:

```
sub ChangeAutopilotAltitudeValue(ByRef refvar, increment, offset, size)
    dim temp
    refvar = refvar + increment
    temp = CInt((refvar / 3.28084) * 65536)
    fs.QueueStateData(offset, size, temp)
end sub
```

This procedure is defined by placing code between the *sub* and *end sub* lines. Immediately after the *sub* statement follows the name of the procedure we are defining (*ChangeAutopilotAltitudeValue*), and optionally, any parameters to be passed to the procedure. In this case, we are passing 4 values, named *refvar*, *increment*, *offset* and *size*, which we will be using to calculate the new value required. All parameters passed to a procedure or function like this hold copies of the data held in them, except when the parameter is prefixed in the procedure or function declaration by *ByRef*. In this case, the *ByRef* means that the parameter is not a copy of the data held, but rather represents the actual variable as passed. Hence in the above rotary encoder *OnChange* statement, we are passing *fsvar\_apaltitude* as the *ByRef* parameter meaning that any use of the *refvar* variable in the procedure will refer to the *fsvar\_apaltitude* variable. The other 3 variables passed (*increment*, *offset* and *size*) are not *ByRef* and therefore are individual variables in their own right within the procedure, and hold copies of the data passed to them which remain individually accessible, and changeable, from the original.

Analysing what happens within the procedure:

**dim temp**

[declares a local variable to this procedure, named *temp*, to act as a temporary store for some later calculations]

**refvar = refvar + increment**

[updates *refvar* with the value of *increment*, which is passed to the procedure in this example as *This.Element.ValueChange\*100*. *ValueChange* holds the encoder's change in value (see earlier explanations). This value is multiplied by 100 to get changes in altitude of 100 feet blocks (changes of single 1 foot increments are not usual for autopilot altitude value settings). *refvar* (which is a *ByRef* pointer to the global variable of *fsvar\_apaltitude*) now holds the new value.]

**temp = CInt((refvar / 3.28084) \* 65536)**

[this converts the new value back to FS units ready to be written to FSUIPC and stores the result in the previously declared *temp* variable]

**fs.QueueStateData(offset, size, temp)**

[this queues a write request to FSUIPC, specifying the offset to write to, the size of the write, and the value to write. In this case we are calling the procedure with *offset* set to "0x07D4" and *size* set to "4" bytes. The *QueueStateData* function allows write requests to be queued and only actioned when *BeginRequest* and *SendRequest* are next called. In this case, this happens constantly during the loop procedure and so the new value will be actioned by FSUIPC and written to the simulator then. Advanced users: *By not explicitly calling BeginRequest and SendRequest within this procedure itself, we can keep the changes to the simulator value very*

*quick by simply updating our global variable with the new value and queuing it for later actioning. If BeginRequest and SendRequest were to be called in this procedure directly, then there would be potentially a large number of these requests to FSUIPC during each encoder detent change, especially if the encoder was to be turned fast. As each BeginRequest and SendRequest takes a few milliseconds to execute, there would be the potential for large delays between changing the encoder and seeing the change within the simulator environment, especially when this effect is multiplied over multiple encoders being twisted at the same time.]*

*Advanced users: You will note that this example uses our global variable "fsvar\_apaltitude" to represent the latest value of the simulator item we want to change. You will also note that we do not read the value from the simulator during each encoder "OnChange" event, as to do so would cause incredible delays between reading, modifying and writing back. It would also cause synchronization errors when the encoder is turned fast as the previous write request would not have had time to become effective in the simulator before the new read request is made. By using a global variable, keeping it updated every so often, and modifying and queuing the value for writing to the FSUIPC in a timely loop, we can avoid all these issues and maintain a very fast encoder-to-simulator link. A useful by-product of this method is that we can also use the value held in the global variable to output to a 7-segment display block, again keeping all changes to such an output very quick.*

## Controlling output nodes - basic LED example

Output nodes can be controlled by simply writing to their *State* property as follows:

Syntax:

```
<the output module name>.Output[<node number>].State = <value>
```

where *<value>* is either 0 (or FALSE) for off, or 1 (or TRUE) for on.

Example:

In the *Loop* section:

```
MyOutputModule.Output[1].State = 1  
MyOutputModule.Output[2].State = 0
```

The above code simply turns on node 1 and turns off node 2.

An example linked to FSUIPC:

In the *General Declarations* section:

```
Dim fs = new ActiveXObject("SBMSFS.FSUIPC")  
Dim fsvar_pause = 0  
fs.Init
```

The above code defines the link to FSUIPC and declares a global variable to hold the FS paused state.

In the *Loop* section:

```
fs.BeginRequest  
fs.ReadStateDataNew(0x0262, 2, GetPtr(fsvar_pause, 3))  
fs.SendRequest  
MyOutputModule.Output[1].State = fsvar_pause
```

This code constantly reads the value of offset 0x0262 from FSUIPC (which represents a value for the paused state of FS, 0 being for unpaused and 1 being for paused). The last line outputs that state to the output node number 1 of *MyOutputModule*.

Additionally, the *State* property of an output node can be assigned as a boolean expression, such as:

```
fs.BeginRequest  
fs.ReadStateDataNew(0x0262, 2, GetPtr(fsvar_pause, 3))  
fs.ReadStateDataNew(0x0BC8, 2, GetPtr(fsvar_parkingbrake, 3))  
fs.SendRequest  
MyOutputModule.Output[1].State = fsvar_pause  
MyOutputModule.Output[2].State = fsvar_parkingbrake = 32767
```

The above shows that the *State* property of node 2 is set to compare the value in *fsvar\_parkingbrake* with the value of 32767. If it matches, the result is TRUE and the output node is switched on. If it does not match, then the result is FALSE and the node is turned off. The FSUIPC Programmer's Guide shows that, for offset 0x0BC8 representing the parking brake, the value returned is 32767 for parking brake set, and 0 for parking brake released. Hence we must use this boolean expression to correctly set the node state.

If we wanted to turn on the node when the parking brake was released rather than set, we could either select the *Invert?* item to *Yes* in the SUC software for this node, or we could do it programmatically, like this:

```
MyOutputModule.Output[2].State = not (fsvar_parkingbrake = 32767)
```

Here, the *not* part inverts the boolean result (TRUE becomes FALSE and FALSE becomes TRUE), thus giving us the desired result of lighting the node when the parking brake is released. Of course, the same effect could be achieved like this:

```
MyOutputModule.Output[2].State = fsvar_parkingbrake = 0
```

## Setting 7-segment display digits

Digit display blocks must be defined as normal in the SUC software, and their justification and leading zeros properties set as desired. Control of the value displayed on the digits is as follows:

In the *Loop* section:

```
MyDigitModule.BaseDigit[1].Poke.Int(123)
```

This code simply sets an integer value of “123” onto the digit block. The *BaseDigit* node number of “1” is the starting digit of the block defined in the SUC software, hence in this case, we must have a block defined of 3 digits starting at digit 1.

Optionally, a format string can also be specified to format the output of the number of the digit block:

```
MyDigitModule.BaseDigit[1].Poke.Int(myVariable, "%-3.3d")
```

In this case, the value held in *myVariable* is set on the block, formatted to “%-3.3d”. The “-” means the formatting is left-justified; the first “3” is the width specifier, the “.” is the precision specifier meaning the result of the format must be at least the specified number of digits long (the 2<sup>nd</sup> “3” in this case) and the “d” means it applies to an integer variable.

If the variable being outputted is a floating point type (a number with a fractional part) then the following can be used:

```
MyDigitModule.BaseDigit[1].Poke.Float(myFloatVariable, "%.3f")
```

In this case, note the *Poke.Float* function is being called, and the formatting is set to have a precision of 3 decimal places.

A string variable can also be output:

```
MyDigitModule.BaseDigit[1].Poke.Str("Std")
MyDigitModule.BaseDigit[4].Poke.Str("345")
```

The above would output “Std” on a block of 3 digits, as defined in the custom characters definition window accessed by right-clicking on a digit node’s green circle in the SUC software. Invalid or undefined characters are ignored. A fixed string of numbers can also be output this way too, as above.

Finally, the output of digits can be controlled by a procedure or function:

```
MyDigitModule.BaseDigit[1].Poke.Code = SetTheDigitsToMyValueByCode
```

with the procedure being defined in the *General Declarations* section:

```
sub SetTheDigitsToMyValueByCode
    dim valueToSet = 123
    This.BaseDigit[This.Node].Int(valueToSet)
end sub
```

Note that we are setting the value on the digits directly within the procedure, and referencing the *This* variable to do so, rather than hard-coding the module and node number.

A digit block can be quickly blanked by using the *Poke.Clear* method:

```
MyDigitModule.BaseDigit[1].Poke.Clear
```

This simply blanks the entire digit block defined from the *BaseDigit* number. This can be useful if you want to simulate blanking of autopilot or radio readouts when no simulated power is available to the aircraft.

## Advanced segment setting - Direct setting of digit segments

Each segment of a 7-segment display digit can be individually set as required. For example, to set the 3 horizontal bars of a digit, you can use this call:

```
MyDigitModule.BaseDigit[1].PokeSegs[3] = 73
```

This code sets digit 3 of the digit block starting at module digit 1 to light the 3 horizontal bars of that digit. 73 represents the value of the 3 bits of the segment, starting at bit 0 as the top horizontal bar of the digit, going clockwise around the outside of the digit, with bit 6 being the middle bar and bit 7 the decimal point.

Bit 0 = 1  
Bit 1 = 2  
Bit 2 = 4  
Bit 3 = 8  
Bit 4 = 16  
Bit 5 = 32  
Bit 6 = 64  
Bit 7 = 128

Hence the 3 horizontal bars we want to light in this example are bits 0, 3 and 6, giving a total value of  $(1+8+64) = 73$ .

If you wanted to set all 3 digits of this block to have the same pattern, you must write each *PokeSegs* item for the block:

```
MyDigitModule.BaseDigit[1].PokeSegs[1] = 73  
MyDigitModule.BaseDigit[1].PokeSegs[2] = 73  
MyDigitModule.BaseDigit[1].PokeSegs[3] = 73
```

## Reading and Writing Bits instead of Bytes

Sometimes it is useful to be able read and write “bits” of data, particularly so when working with PM Systems and some other 3rd-party programs. A bit is a component of a byte, there being 8 bits in 1 byte. Each bit holds an “on” or “off” state (1 or 0, or TRUE or FALSE). Since the smallest variable structure we can pass to FSUIPC to be read or written is 1 byte, it follows that to read a given bit of a byte (or write back a given bit of a byte), we must first work with the byte to which the required bit relates. After doing so, we can use the value of that bit, modify it or use it for output, and if necessary, write it back as a byte (the 7 other bits of the byte remaining unchanged).

An example:

In the *General Declarations* section:

```
Dim fs = new ActiveXObject("SBMSFS.FSUIPC")
fs.Init
Dim fsvar_56E5
```

In the *Loop* section:

```
fs.BeginRequest
fs.ReadStateDataNew(0x56E5, 1, GetPtr(fsvar_56E5, 3))
fs.SendRequest
MyOutputModule.Output[1].State = IsBitSet(fsvar_56E5, 3)
MyOutputModule.Output[6].State = IsBitSet(fsvar_56E5, 2)
```

In this example, all we are doing is reading the byte at offset 0x56E5, which is the PMSystems (737NG) offset that holds the states of each of the 4 Window Heats in 4 separate bits of that byte. Specifically, bit 0 holds the state of the “Left Side” window (on or off), bit 1 the “Left Forward” window, bit 2 the “Right Forward” window and bit 3 the “Right Side” window. The remaining 4 bits (bits 4 to 7) of the byte are not used by PM. An explanation of the code above follows:

**MyOutputModule.Output[1].State = IsBitSet(fsvar\_56E5, 3)**

[this line simply sets the state of output node 1 to TRUE if the 3<sup>rd</sup> bit (bit 3) of the value held in fsvar\_56E5 variable is set (that is, the “IsBitSet” function returns TRUE or FALSE). In much the same manner, node 6 is set to light up when bit 2 is set (representing the “Right Forward” window heat being on).]

Another example:

In this second example, we will read a number of bytes associated with a whole group of on/off conditions within PM. The example we will use is 0x04F0 for 2 bytes this time (meaning we are actually reading 0x04F0 and 0x04F1), totalling 16 bits (over 2 bytes):

In the *General Declarations* section:

```
Dim fs = new ActiveXObject("SBMSFS.FSUIPC")
fs.Init
Dim fsvar_737NG_AP_bits
```

In the *Loop* section:

```
fs.BeginRequest
fs.ReadStateDataNew(0x04F0, 2, GetPtr(fsvar_737NG_AP_bits, 3))
fs.SendRequest
MyOutputModule.Output[1].State = IsBitSet(fsvar_737NG_AP_bits, 0)
```

```
MyOutputModule.Output[2].State = IsBitSet(fsvar_737NG_AP_bits, 6)
MyOutputModule.Output[3].State = IsBitSet(fsvar_737NG_AP_bits, 14)
```

This time, the *ReadStateDataNew* request is made specifying that we want to read from offset 0x04F0 (1<sup>st</sup> parameter), for 2 bytes (2<sup>nd</sup> parameter) and hold the result in a pointer to our variable called *fsvar\_737NG\_AP\_bits*. We then check bits 0, 6 and 14 of that structure, and assign the TRUE / FALSE state returned by the check to nodes 1, 2 and 3 of an output module. In this case, node 1 will light when the AP Master Left is on; node 2 when LNAV is engaged; and node 3 when VNAV is engaged.

## Using a switch to modify a bit parameter in PMSystems

A lot of switch states in PMSystems are held in FSUIPC offset locations as bits. If we want to control a given bit within PMSystems when one of our hardware switches is moved, we must first read the existing byte from FSUIPC that contains the bit we want to modify, change the bit to reflect the new status of our switch, and then write back the byte (remember, this holds 8 bits) with just the bit in question modified (the remaining 7 bits must remain unchanged, otherwise 7 other switch states may be incorrectly changed too).

To do this, an example:

In the *General Declarations* section:

```
Dim fs = new ActiveXObject("SBMSFS.FSUIPC")
fs.Init
Dim fsvar_WindowHeat_SwitchBits
```

In the *Loop* section:

```
fs.BeginRequest
fs.ReadStateDataNew(0x56E4, 1, GetPtr(fsvar_WindowHeat_SwitchBits, 3))
fs.SendRequest
```

Also in the *General Declarations* section:

```
MyInputModule.Input[1].OnStateChange = "fs.BeginRequest; fs.WriteStateData(0x56E4, 1,
BitOut(fsvar_WindowHeat_SwitchBits, 0, This.Element.State)); fs.SendRequest"
```

Here, we set up our FSUIPC link as normal, and specify some code to execute when the switch node 1 changes on our input module. Additionally, and crucially, we are also using the Loop section to ensure that we always have a variable being filled with the contents of the byte at offset 0x56E4, which in PMSystems holds the switch positions for each of the 4 window heats in the first 4 bits (bit 0 to bit 3) of that byte. That way, whenever we flick our switch all we have to do is refer to our constantly-updated variable, modify the required bit of that variable (thereby leaving the remaining 7 bits of the variable unmodified), and write the variable back to FSUIPC as a byte to fill the offset 0x56E4 again. PMSystems will pick up on the change made, and effect the change in the PMSystems software to change the state of the software switch.

The *OnStateChange* code line is similar to the very basic input example towards the start of this document. However, an explanation would be useful in this case.

```
fs.WriteStateData(0x56E4, 1, BitOut(fsvar_WindowHeat_SwitchBits, 0,
This.Element.State))
```

[this part of the code specified to execute when the switch changes state does 2 things: firstly, it is calling the *WriteStateData* function to write out "1" byte of data (2<sup>nd</sup> parameter) to offset

“0x56E4” (1<sup>st</sup> parameter), that byte of data being the value returned by the *BitOut* function. The *BitOut* function returns a variable with a specified bit changed, and takes the following parameters: the variable to use (1<sup>st</sup> parameter, and in our case, is the value held in *fsvar\_WindowHeat\_SwitchBits*); the bit number to change (2<sup>nd</sup> parameter, and in our case we are specifying to change bit “0” which represents the position of the “Left Side” window heat switch); and the new value for the specified bit (3<sup>rd</sup> parameter, and in our case is the new state of the switch, on/true or off/false).]

After changing our switch position, the code will execute and write back to FSUIPC a new byte containing the bit 0 modified to the actual value of our hardware switch. Once PMSystems has processed this request, further reads of this offset during the Loop procedure will reflect the new PMSystems software state of this switch too.

## Using Include files

Include files are simple: they allow you to create your project from one or more files that may hold specific code. For example, the “FS\_IMPORTS.txt” file, located in your SIM-board installation under the “Libraries” folder, contains a function named *FSWrite* which encapsulates the *BeginRequest...WriteStateData...SendRequest* sequence, thereby relieving you of having to type this sequence into every node with which you want to write a change back to FSUIPC. Referring to the first switch example towards the top of this document, the code could be abbreviated to:

In the *General Declarations* section:

```
Include("FS_IMPORTS.txt")  
MyInputModule.Input[1].OnStateOn = "FSWrite(0x0BC8, 2, 1)"
```

You can see this is much clearer to read, and makes good use of a pre-defined procedure held in an include file. The statement *Include("<name of file">)* is used to insert the file at that location when the project is run, and must refer to a file in the “Libraries” folder of your SIM-board installation.

## Using potentiometers

Code for accessing the present position of potentiometers is able to be used to control certain functions too. For example, we might want to use a potentiometer to set the thrust lever position in Flight Simulator. Use the SIM-board software to set up your potentiometer and calibrate it as normal, and then use something similar to the following code:

In the *General Declarations* section:

```
MyInputModule.Pot[1].OnStateChange = "fs.QueueStateData(0x088C, 2, Cint((This.Element.ValuePC/100)*16383))"
```

In the *Loop* section:

```
fs.BeginRequest  
fs.SendRequest
```

This code very simply sets up pot node 1 on our input module to execute the *fs.QueueStateData* function when the pot changes value. Additionally, we establish a constant *BeginRequest...SendRequest* loop to FSUIPC in the loop section. Let's look now at the *QueueStateData* request and what we are asking it to do:

```
fs.QueueStateData(0x088C, 2, Cint((This.Element.ValuePC/100)*16383))
```

[here, the *QueueStateData* function acts similarly to the *WriteStateData* function, the difference being that the request is queued until the next *SendRequest* is processed. This allows us to queue requests for sending to FSUIPC, rather than calling *BeginRequest...WriteStateData...SendRequest* every time our pot changes value (which could be many times per second, and cause considerable processing delays). The *QueueStateData* function itself takes 3 parameters: the offset to write to (1<sup>st</sup> parameter, and in this case is 0x088C which is the engine 1 thrust lever position control offset); the number of bytes to write (2<sup>nd</sup> parameter, and "2" in this case); and the value to write to that offset (3<sup>rd</sup> parameter). This example shows that we want to write a value based on a mathematical calculation using the *ValuePC* property of the pot node. *ValuePC* represents the value of the pot across its calibrated range, expressed as a percentage from 0 to 100. Hence, the calculation above takes this *ValuePC* value and divides it by 100, to give a position of between 0.0 and 1.0, and multiplies that by 16383, which is the value in FSUIPC that represents 100% thrust lever position (fully forward). The *Cint* part simply converts the result of that mathematical calculation into an integer number (a number without a fractional part) that is suitable from writing back to FSUIPC at that offset. Refer to the FSUIPC Programmer's Guide for more information on the mathematical conversions required for various offsets.]

Additionally, pot values can be read giving their raw position data in 10-bit resolution (ie. values of 0 to 1023). Use *MyInputModule.Pot[1].Value* (or *This.Element.Value*) to get this. Most of the time *ValuePC* will probably be more useful.

## Controlling a potentiometer configured as a multiple zone "Pot as Switch" type

When your pot is set up to be a multiple zone type in the SUC software, you can specify code to be executed when the pot moves into a zone, out of a zone, and when it moves within a zone. The first two cases are covered by the *OnStateEnter* and *OnStateExit* properties of the pot, and the latter by the *OnStateMove* property. An example follows:

In the *General Declarations* section:

```
PedestalInputs.Pot[4].Zone[1].OnStateEnter = "fs.QueueStateData(0x088C, 2, -4096)"
PedestalInputs.Pot[4].Zone[2].OnStateEnter = "fs.QueueStateData(0x088C, 2, 0)"
PedestalInputs.Pot[4].Zone[3].OnStateMove = "fs.QueueStateData(0x088C, 2,
Cint((This.Element.ValuePC/100)*16383))"
```

Note here that the first 2 zones are used to set absolute values of thrust lever position (zone 1 sets a value of -4096 which represents full reverse thrust, hence the zone will have been configured in the SUC software to represent this reverse thrust range of movement on our physical thrust lever). Zone 2 is also a fixed zone and writes 0 to the thrust lever offset (meaning the idle position). Zone 3 is the forward thrust zone of movement for the thrust lever, and the reference to *This.Element.ValuePC* holds the percentage position of the pot within the zone itself (not the pot as a whole). Hence *ValuePC* will be 0 at the very start of zone 3 and 100 at the very end of zone 3.

Note that you can use all 3 events (*OnStateEnter*, *OnStateExit*, *OnStateMove*) for a zone, if you need to. You are not restricted to using just one. For example, you might want to execute some code when the pot moves into the zone and also when it exits the zone.

## Sending Keystrokes

To send keystrokes to the active window on the PC running the SUC software, you can use the *SendKeys* function. It operates like this:

```
SendKeys ("+(hd) ")
```

The above example passes one string as the only parameter to the *SendKeys* function. The string itself consists of:

```
+(hd)
```

which means press the “h” key then the “d” key, all while the shift key is held down (represented by the “+”). The brackets around the “hd” part signify that the shift key should be held down throughout the “hd” operation, and then released at the end of the bracket.

Another example:

```
SendKeys ("+(hd) ^ ( {F3} ) % ( {DEL} ) ")
```

This example performs “shift down” then “h” then “d” then “shift release”, followed by “ctrl down” (the “^” symbol) and the “F3” key (the names of these function keys are all enclosed in curly brackets) and then “ctrl release”, followed by “alt down” (the “%” symbol) then the “DEL” key then “alt release”. This is an extreme example but covers a number of different possibilities.

Other keys that are usable in curly brackets include:

BACKSPACE	ESCAPE	F10	LEFT
BREAK	F1	F11	NUMLOCK
CAPSLOCK	F2	F12	PGDN
CLEAR	F3	F13	PGUP
CTRL	F4	F14	PRTSC
DEL	F5	F15	RIGHT
DELETE	F6	F16	SCROLLLOCK
DOWN	F7	HELP	SHIFT
END	F8	HOME	TAB
ENTER	F9	INS	UP

Optionally, a key repeat can be established. The repeat interval is passed as the 2<sup>nd</sup> parameter to *SendKeys*, in milliseconds. The key repeat will continue until cancelled by another call to *SendKeys*, most likely in the *OnStateOff* event of a switch node, by passing the same string of key data as the 1<sup>st</sup> parameter, and “-1” as the 2<sup>nd</sup> parameter. This tells the software to cancel the key repeat for that specific string while keeping any other key repeats that happen to be active. An example follows:

```
InputModule.Input[28].OnStateOn = "SendKeys ("+(s) ", 500) "  
InputModule.Input[28].OnStateOff = "SendKeys ("+(s) ", -1) "
```

Firstly, note that when used in the context of assignment to *OnState* events of nodes that require their event code to be contained within quotation marks (“”), we must double-quote the first parameter of the *SendKeys* function. If you do not, the *OnStateOn* and *OnStateOff* definitions in the example above will be incorrect and your code will not run.

Secondly, you can see that the keys we are wanting to simulate in this instance are “Shift” then “s” then “shift release”. The 2<sup>nd</sup> parameter is “500” meaning that we want to perform this key sequence every 500ms (half a second). Note that we also define the *OnStateOff* event to ensure that this key repeat sequence is disabled when the switch goes off. If we did not do this, the

key repeat would remain active, executing every 500ms, and would never get cancelled during the running of the project. So, by assigning *OnStateOff* with the same key sequence but passing “-1” as the 2<sup>nd</sup> parameter, we can cancel this key sequence.

Of course, you don't have to use *SendKeys* in the *OnState* events of input nodes. You could, if you wanted, simply call *SendKeys* like any other function. An example follows:

```
MyInputModule.Input[8].OnStateChange = "CheckSwitchAndDoSomething(This.Element.State)"
```

```
sub CheckSwitchAndDoSomething(switchIsOnQ)
  if switchIsOnQ then
    SendKeys("yes")
  else
    SendKeys("no")
  end if
end sub
```

This simple example calls our own procedure named *CheckSwitchAndDoSomething* and passes the current state of the switch as the only parameter. In the procedure (sub) itself, the passed parameter is checked and if it is TRUE (that is, the switch is on) then we use *SendKeys* to type the word “yes”. Otherwise the switch is FALSE (off) and we type “no”. Running this project with Notepad open as the active window, and then toggling the switch, should produce the necessary output in the Notepad window.

## Importing 3rd-party DLL functions into your project

It is often useful, particularly when you are wanting to interface to a 3rd-party application, to import DLL functions that can access variables of the other program (for example, you might have a transponder program that simulates the aircraft transponder, and you want to read the squawk code digits from it and also notify it when you press the ident button on your hardware). To do this, the application programmer could use an ActiveX DLL and access it in much the same manner as already described for the SBMSFS.DLL supplied with the SUC software to aid access to FSUIPC, or the programmer could provide a normal DLL that exports some functions that we can call from our SSL script project to read and write data.

An example might be:

In the *General Declarations* section:

```
Dim DllName = "Transponder.dll"  
Declare Function GetTransponderCode Lib DllName As Integer  
Declare Function IdentButtonPressed Lib DllName As Byte  
Declare Function SetTransponderCode Lib DllName (NewCode As Integer) As Byte
```

The code above declares 3 functions to be imported into our project from a DLL named "Transponder.dll". The script will attempt to load the DLL from the SIMboards.exe path in this case; if you want to specify an exact path, you can do so as well. The functions might be used in subsequent SUC code like this:

In the *General Declarations* section:

```
MyInputModule.Input[1].OnStateOn = "IdentButtonPressed"  
MyInputModule.Rotary[5].OnChange = "SetTransponderCode(GetTransponderCode +  
This.Element.ValueChange)"
```

Note that you can, in fact, import both functions and procedures (subs) from a 3rd-party DLL, but that they must all be declared as *Functions* in your script with a return result (even though a procedure by definition has no return result). So in fact, *IdentButtonPressed* and *SetTransponderCode* in the above example could actually be procedures, not functions, within the DLL (but we have to declare them using *Declare Function* and with *As Byte* as the default return type in script).

The default calling convention is *stdcall*; if you need to import functions that use different calling conventions, then use *Declare <convention> Function ....* instead:

```
Declare cdecl Function ExampleFunctionFromDLL Lib DllName As Integer  
Declare register Function AnotherExampleFunctionFromDLL Lib DllName As Integer
```

Here the calling conventions used are *cdecl* and *register*, respectively. Ensure you use the correct calling convention for each exported function as used in the host DLL.

## Input repeater events

In some cases, you will want to repeat the execution of code once a switch is in a certain position. Similarly, you will want to cancel the repeat execution of that code when something else happens (maybe when the switch moves to another position). An example is the electric trim switch on the control wheel of a Boeing 737: when the switch is moved to the up position, nose down trim should be applied at a certain rate, and even though the switch position is not changing, the trim should keep being applied at that rate until the switch is moved back to the off (central) position. The library file for FSUIPC and the “FS\_IMPORTS.txt” file contains a good, commented example of doing this. As an easier example of a basic input repeater event, refer to the below:

In the *General Declarations* section:

```
Dim myCounter = 0

MyInputModule.Input[1].OnStateOn = "This.Element.RepeatEvent(TRUE, "myCounter = myCounter + 1; debugprint Str(myCounter)")"

MyInputModule.Input[1].OnStateOff = "This.Element.RepeatEvent(FALSE, "myCounter = myCounter + 1; debugprint Str(myCounter)")"
```

Note that we have defined 2 state events, one for the switch going on and the other for when it goes off. When it goes on, we are asking for the *RepeatEvent* to be set up (a value of TRUE in the 1<sup>st</sup> parameter of the *RepeatEvent* function), and to perform the code in double quotes every second (2<sup>nd</sup> parameter). The code itself increments our internal counter and simply outputs it to the debug tab window of the SUC software. The *OnStateOff* code requests the *RepeatEvent* be cancelled (FALSE) and refers to the exact same code line in order that the software can properly identify the repeat event to cancel.

## SIM-board Scripting Language: Useful SIM-board functions

Timer As Integer	Returns integer from GetTickCount Windows function.
MouseClicked(X, Y As Integer)	Performs a simulated mouse click on the screen at the specified coordinates.
HitKey(KeyCode As Integer)	Performs a basic press-and-release of the specified keycode.
PressKey(KeyCode As Integer)	Performs a basic press of the specified keycode, with no release.
ReleaseKey(KeyCode As Integer)	Performs the release of a previously held key from PressKey.
Min(A, B As Integer) As Integer	Returns the minimum value of the two variables passed.
Max(A, B As Integer) As Integer	Returns the maximum value of the two variables passed.
Val(theString As String)	Returns the integer, or float value of a string passed. If the string is a hex representation (eg. "&H04FD") then the integer value of that hex value is returned.
Swap(ByRef A As Variant, ByRef B As Variant)	Swaps the two passed parameters in value.
BitSet(ByRef V As Integer, setOn As Boolean, TheBit As Byte)	Procedure sets TheBit bit to setOn state in the V variable passed.
BitToggle(V As Integer, TheBit As Byte) As Integer	Returns the value of V modified with the bit TheBit toggled in state.
BitOut(V As Integer, TheBit As Byte, setOn As Boolean) As Integer	Returns the value of V modified with TheBit bit changed to setOn state passed.
IsBitSet(V As Integer, TheBit As Byte) As Boolean	Returns TRUE when TheBit bit of variable V is set, otherwise returns FALSE.
HighByte(V As Integer) As Byte	Returns the high value byte of variable V.
LowByte(V As Integer) As Byte	Returns the low value byte of variable V.
CopyMemory(ByVar Destination As Variant, Source As Variant, Size As Integer)	An implementation of the Windows CopyMemory function.
DoEvents	Forces execution of pending Windows messages within the program. Similar to DoEvents in VisualBasic.
Format(Format As String, Args[] As Array of Variant) As String	Returns a string formatted according to the Format and Args passed.
DebugPrint(Msg As String)	Outputs the Msg string passed to the Debug tab area of the SIM-board Universal Controller software main window.
Include(TheFile As String)	Includes the specified file, which must be located in the "Libraries" folder of the SUC installation, at this location in the General Declarations section. Not valid in any other section.

Beep	Triggers a beep sound from Windows.
SIMboards.Ready As Boolean	Returns TRUE once the SIM-boards are first powered up and booted and the main program loop is executing.
SIMboards.AllAlive As Boolean	Returns TRUE during program execution when all USB chain connections are alive and booted, and all modules of each chain are also alive and booted.
<InputModule>.RequestStates	Forces a request to all input modules for current switch states.
<OutputModule>.RefreshOutputs	Forces an update of all output nodes on the module to their current software states as set.
<OutputModule>.RefreshBrightness	Forces an update of all output node brightnesses to their current software brightness value. This procedure is not normally called by user programs but is called automatically each <i>Loop</i> cycle.
<DigitModule>.RefreshDigitPokes	Forces an update of all the digits on the module to reflect their current software values.
<DigitModule>.RefreshDigitBrightnesses	Forces an update of all the digit brightnesses on the module as currently set in software.

## SIM-board Scripting Language: Built-in functions and procedures

The following are some of the SIM-board Scripting Language functions available for use. Many are very similar to their VisualBasic equivalents. Explanations for these functions may be provided in a future version of this document or on user request.

Abs	
Array	
Asc	
Atn	
CBool	
CByte	
CDate	
CDBl	
Chr	
CInt	
CLng	
CSmallInt	
Cos	
CLng	
Date	
DateSerial	
DateValue	
Day	
Eval	
Exp	
Filter	
FormatDateTim e	
FormatNumber	
FormatPercent	
GetObject	
GetRef	
<b>GetPtr</b>	SIM-board function - returns a pointer to the passed variable (1 <sup>st</sup> parameter), typecast to the type index specified in the 2 <sup>nd</sup> parameter. Typecasts available: 2 = SmallInt 3 = Integer (LongInt) 4 = Single 5 = Double 11 = Boolean 17 = Byte
Hex	
Hour	

InputBox
InStr
InStrRev
Int
Fix
IsArray
IsDate
IsEmpty
IsNull
IsNumeric
IsObject
Join
LBound
LCase
Left
Len
Log
LTrim
RTrim
Trim
Mid
Minute
Month
MonthName
MsgBox
Now
Replace
RGB
Right
Rnd
Round
Second
Sgn
Sin
Space
Split
Sqr
StrComp
Str

Tan	
Time	
Timer	
TimeSerial	
TimeValue	
TypeName	
UBound	
UCase	
VarType	
<b>VV</b>	SIM-board function: short for “VerifyVariable”, this takes 2 parameters and returns the value of the first one if it is valid, otherwise it returns the value of the second one.
Weekday	
Year	
Substr	

## Quick Reference

- **Switch nodes**

Use in *General Declarations* section

```
InputModule.Input[1].OnStateOn = "<code to execute>"
InputModule.Input[1].OnStateOff = "<code to execute>"
InputModule.Input[1].OnStateChange = "<code to execute>"
```

Be sure to double-quote any code within the main quotation marks for each of these statements, if it is necessary.

### Switch properties

```
.State
```

- **Rotary Encoder nodes**

Use in *General Declarations* section

```
InputModule.Rotary[1].Idle.Setup(<interval>, TRUE, "<code to execute>")
InputModule.Rotary[1].OnChange = "<code to execute>"
```

At any point in code, if you want to call the *Idle* event directly there and then:

```
InputModule.Rotary[1].Idle.Execute
```

### Rotary Encoder properties

```
.ValueChange
```

- **Potentiometer nodes**

Use in the *General Declarations* section

For pots defined as normal pots:

```
InputModule.Pot[1].OnStateChange = "<code to execute>"
```

For pots defined as multiple-zone pots:

```
InputModule.Pot[1].Zone[1].OnStateEnter = "<code to execute>"
InputModule.Pot[1].Zone[1].OnStateExit = "<code to execute>"
InputModule.Pot[1].Zone[1].OnStateMove = "<code to execute>"
```

### Potentiometer and Zone properties

```
.Value
.ValuePC
```

- **Digit nodes**

Use in the *Loop* section

```
DigitModule.BaseDigit[1].Poke.Clear  
DigitModule.BaseDigit[1].Poke.Int(<value>[, "<optional justification string>"])  
DigitModule.BaseDigit[1].Poke.Float(<value>[, "<optional justification string>"])  
DigitModule.BaseDigit[1].Poke.Str("<value>"[, "<optional justification string>"])  
DigitModule.BaseDigit[1].PokeSegs[1] = <value>
```

- **Output nodes**

Use in the *Loop* section

```
OutputModule.Output[1].State = <value>
```