

# **MaxCompiler**

## **State Machine Tutorial**

Version 2014.2



# Contents

<b>Contents</b>	<b>1</b>
<b>Preface</b>	<b>3</b>
<b>1 Overview</b>	<b>5</b>
<b>2 Kernel State Machine Example</b>	<b>7</b>
2.1 The State Machine . . . . .	7
2.2 The Kernel . . . . .	9
<b>3 State Variables</b>	<b>12</b>
3.1 Value Assignment Operator . . . . .	12
3.2 Enumerated State Variables . . . . .	12
3.3 Value State Variables . . . . .	13
<b>4 Sequential Statements</b>	<b>14</b>
4.1 The Next State Function . . . . .	14
4.2 The Output Function . . . . .	14
4.3 Conditional Statements . . . . .	15
4.3.1 IF statements . . . . .	15
4.3.2 SWITCH statements . . . . .	15
<b>5 Kernel State Machines</b>	<b>17</b>
5.1 Inputs and Outputs . . . . .	17
5.2 Output Latency . . . . .	18
5.3 Input Validity . . . . .	18
<b>6 Manager state machines</b>	<b>19</b>
6.1 Push IO . . . . .	19
6.1.1 Stall Latency . . . . .	19
6.1.2 Push IO Transfer Examples . . . . .	20
6.2 Pull IO . . . . .	22
6.2.1 Almost-Empty Signal . . . . .	22
6.2.2 Pull IO Transfer Examples . . . . .	23
6.3 Pull Input to Push Output Example: Morse Code Tokenizer . . . . .	25
6.3.1 Setting up the Input and Output . . . . .	25
6.3.2 Read Control . . . . .	29
6.3.3 Write Control . . . . .	29
6.3.4 Integrating with a Custom Manager . . . . .	30
6.4 Push Input to Pull Output Example: Sorter . . . . .	31
6.4.1 Setting up the Input and Output . . . . .	31
6.4.2 Read Control . . . . .	36
6.4.3 Write Control . . . . .	37
6.5 Selecting IO Types . . . . .	38
<b>7 Scalar Inputs and Outputs</b>	<b>39</b>
7.1 Accessing Scalar IO in a State Machine . . . . .	39

---

7.2	Accessing Scalar IO From the CPU	42
7.2.1	Kernel State Machines	42
7.2.2	Manager State Machines	42
<b>8</b>	<b>Memories</b>	<b>44</b>
8.1	Memory Latency	44
8.2	ROMs	44
8.2.1	Single-Port ROMs	44
8.2.2	Dual-Port ROMs	45
8.3	RAMs	45
8.3.1	Single-Port RAMs	45
8.3.2	Dual-Port RAMs	45
8.4	Mapped Memories	46
8.4.1	Mapped ROMs	46
8.4.2	Mapped RAMs	47
8.5	Dual-Port Mapped RAM Example	47
8.5.1	The state machine	47
8.5.2	The CPU code	48
<b>9</b>	<b>Debugging</b>	<b>53</b>
9.1	Identifying the State Machine Instance	53
<b>10</b>	<b>Modularizing Code</b>	<b>55</b>
10.1	Intermediate Values	55
10.2	State Machine Libraries	55
10.3	State Machine Library Example	56
10.3.1	The State Machine Library	56
10.3.2	The Kernel State Machine	58
10.3.3	The Manager State Machine	59
<b>11</b>	<b>Exercises</b>	<b>62</b>
11.0.4	Exercise 1: Morse code decoding using a ROM	62
11.0.5	Exercise 2: Run-Length decoding in a Manager state machine	63

---

## Preface

### Purpose of this document

This document is designed to lead the reader through the use of state machines in MaxCompiler, presenting examples where appropriate. Each section introduces a new set of features and has examples showing their use.

### Document Conventions

When important concepts are introduced for the first time, they appear in **bold**. *Italics* are used for emphasis. Directories and commands are displayed in typewriter font. Variable and function names are displayed in typewriter font. Java methods and classes are shown using the following format:

```
DFEsmSinglePortMappedRAM mem.ramMapped(String name, DFEsmValueType type, int depth, SinglePortRAMMode portMode,  
    Latency latency)
```

C function prototypes are similar:

```
void SimpleSM(  
    int32_t param_N,  
    const uint32_t *instream_max,  
    uint32_t *outstream_count);
```

Actual Java usage is shown without italics:

```
m_mode.next <= Modes.HOLD_COUNT;
```

C usage is similarly without italics:

```
ScalarSM(size, holdCount, &maxCount, dataIn, dataOut);
```

Sections of code taken from the source of the examples appear with a border and line numbers:

```
1  /**
2   * Document: MaxCompiler State Machine Tutorial (maxcompiler-sm-tutorial.pdf)
3   * Example: 1      Name: Simple statemachine
4   * MaxFile name: SimpleSM
5   * Summary:
6   *   A simple kernel that send inputs from "max" to the state machine
7   *   and sends the output of this statemachine to count.
8   */
9  package simplesm;
10
11  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
12  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
13  import com.maxeler.maxcompiler.v2.kernelcompiler.SMIO;
14  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
15
16  class SimpleSMKernel extends Kernel {
17
18      private static final int width = 32;
19
20      SimpleSMKernel(KernelParameters parameters, int holdCount) {
21          super(parameters);
22
23          DFEVar max = io.input("max", dfeUInt(32));
24
25          SMIO mySimpleSM = addStateMachine("SimpleSM", new SimpleSMStateMachine(
26              this, width, holdCount));
27
28          mySimpleSM.connectInput("max", max);
29
30          DFEVar count = mySimpleSM.getOutput("count");
31
32          io.output("count", count, count.getType());
33      }
34  }
```

## 1 Overview

State machines in MaxCompiler allow you to implement fine-grained control blocks to control the flow of data in a streaming Kernel or Manager. State machines offer low-level control with a more direct mapping to hardware than a streaming Kernel.

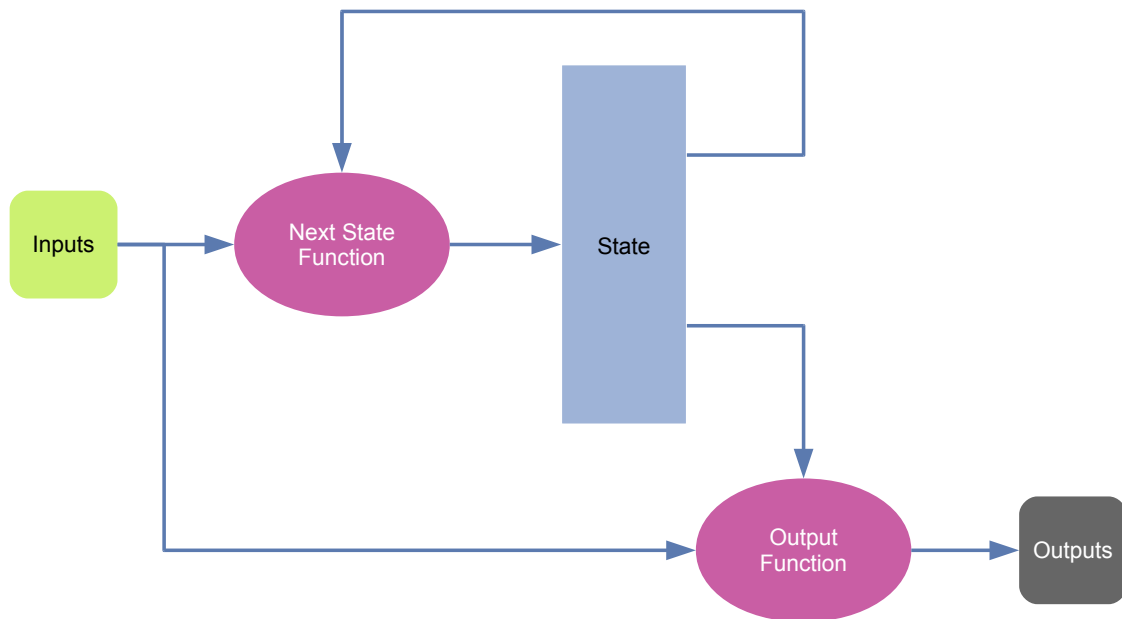


Figure 1: The state machine model.

Figure 1 is a graphical representation of the state machine model used in MaxCompiler. Input and output streams are connected to the state machine.

The behavior of the state machine each cycle is described by two small programs, the *Next State Function* and the *Output Function*. The Next State Function calculates the next state from the current state and the current inputs. The Output Function sets the output based on the current state and the current inputs.



The term “state” refers to all data that is preserved from one cycle to the next.

The Next State Function and Output Function are fully executed once, in parallel with each other, every cycle and each takes a single cycle to complete. The next state is stored and becomes the current state in the following cycle. Figure 2 shows the relationship between the state, inputs and outputs for a given cycle,  $t$ .

There are two use cases for a state machine block:

- In a Kernel. When implementing complex control flow, a state machine block can be easier to write and maintain than using counters and control streams.
- In a Manager. Sometimes the streaming model is not appropriate: a state machine block allows you to implement state machine logic and integrate it seamlessly into the streaming model of the Manager.

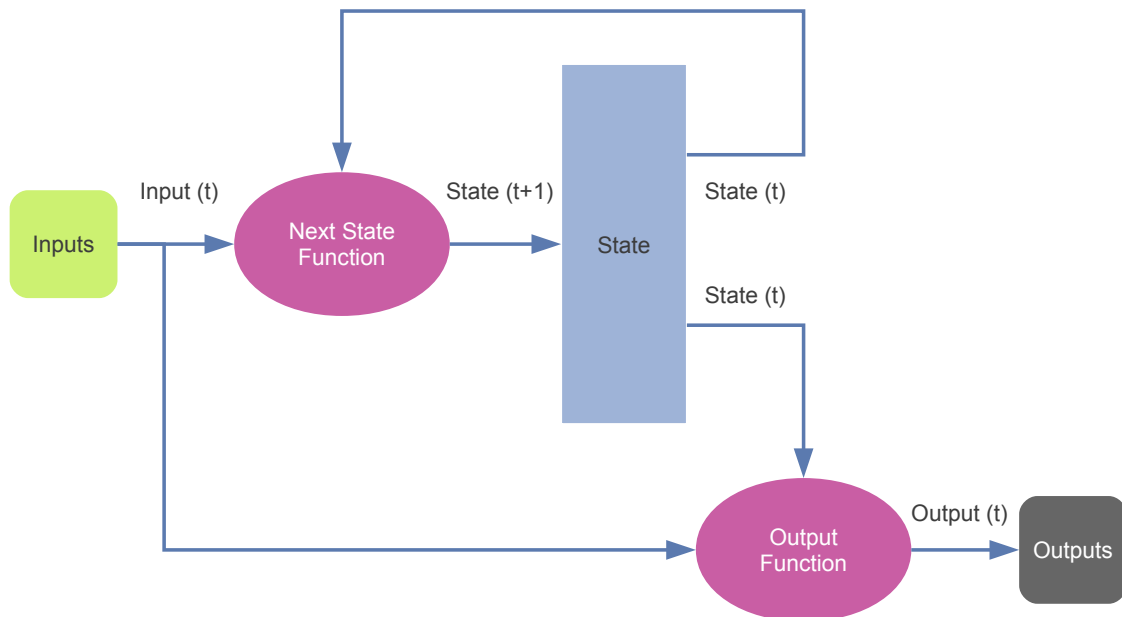


Figure 2: State machine model showing the relationship between state, inputs and outputs for cycle  $t$ .

Kernel and Manager state machines follow the same basic execution model described above, with two key differences:

- A Kernel state machine is executed every tick of the Kernel in which it is instantiated; a Manager state machine is executed every clock cycle.
- A Kernel state machine receives a new data item on every input every cycle and produces an output every cycle; a Manager state machine must implement a protocol on each input and output to send and receive data.

## 2 Kernel State Machine Example

In this section, we will follow the simple Kernel state machine in Example 1. The source code for the state machine for this example is shown in [Listing 1](#) and [Listing 2](#).

### 2.1 The State Machine

The example has one input and one output. The output (`count`) is a counter that counts from 0 up to the maximum value specified by the input (`max`), at which point it holds the value for a number of cycles before it counts back down to 0 again. It then repeats the cycle. It outputs a new value for the counter every cycle. If the current counter value is larger than the current maximum value from the input stream, then the counter will be set to this maximum value the following cycle and held. [Figure 3](#) shows the MaxCompiler state machine model with the variable and function names from Example 1 annotated.

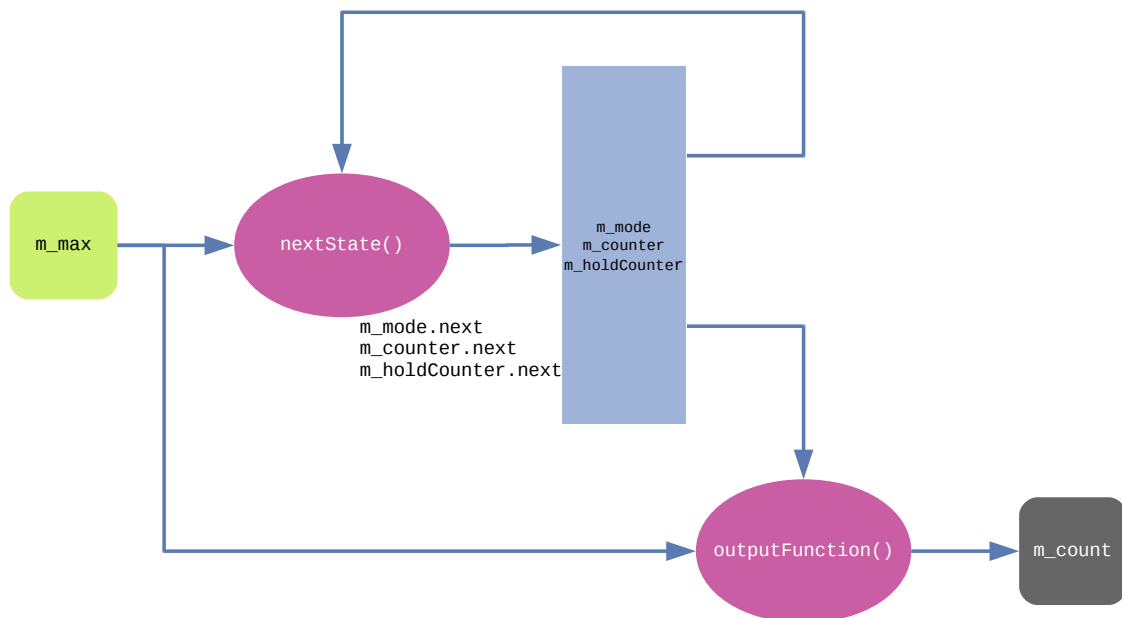


Figure 3: The state machine model with the function and variable names annotated.

The code for a state machine block for use in a Kernel is implemented in a class that extends `KernelStateMachine`:

```
17 class SimpleSMStateMachine extends KernelStateMachine {
```

The inputs, outputs and state variables are declared as class fields:

```
22 // I/O
23 private final DFESmOutput oCount;
24 private final DFESmInput iMax;
25
26 private final int mHoldCount;
27
28 // State
29 private final DFESmStateValue sCounter;
30 private final DFESmStateValue sHoldCounter;
31 private final DFESmStateEnum<Modes> sMode;
```



The initial state is typically defined in the constructor. The input and output variables are also connected to the named input and output in the constructor:

```

33 public SimpleSMStateMachine(KernelLib owner, int width, int holdCount) {
34     super(owner);
35     mHoldCount = holdCount;
36     DFEsmValueType counterType = dfeUInt(width);
37     // I/O
38     oCount = io.output("count", counterType);
39     iMax = io.input("max", counterType);
40     // State
41     sMode = state.enumerated(Modes.class, Modes.COUNTING_UP);
42     sCounter = state.value(counterType, 0);
43     sHoldCounter = state.value(counterType, 0);
44 }

```

Within the SimpleSMStateMachine class, the abstract methods outputFunction() and nextState() must be implemented. The Next State Function and Output Function are *fully* executed once, in parallel with each other, every cycle.

The Next State Function (nextState()) contains the logic to update the state from one cycle to the next:

```

46 protected void nextState() {
47     SWITCH(sMode) {
48         CASE(Modes.COUNTING_UP) {
49             IF(sCounter == iMax) {
50                 sMode.next <= Modes.HOLD_COUNT;
51             } ELSE {
52                 /* We need to check that we are not larger than
53                  * m_max, as it might have changed
54                  */
55                 IF(sCounter > iMax) {
56                     sCounter.next <= iMax;
57                     sMode.next <= Modes.HOLD_COUNT;
58                 } ELSE {
59                     sCounter.next <= sCounter + 1;
60                 }
61             }
62         }
63         CASE(Modes.COUNTING_DOWN) {
64             IF(sCounter == 0) {
65                 sCounter.next <= sCounter + 1;
66                 sMode.next <= Modes.COUNTING_UP;
67             } ELSE {
68                 sCounter.next <= sCounter - 1;
69             }
70         }
71         OTHERWISE {
72             IF (sHoldCounter == mHoldCount)
73             {
74                 sHoldCounter.next <= 0;
75                 sMode.next <= Modes.COUNTING_DOWN;
76             }
77             ELSE {
78                 sHoldCounter.next <= sHoldCounter+1;
79             }
80         }
81     }
82 }

```

## 2 Kernel State Machine Example

*Listing 1: Program for the simple state machine (Part 1 of SimpleSMStateMachine.maxj).*

```

8 package simplesm;
9 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelLib;
10 import com.maxeler.maxcompiler.v2.statemachine.DFEsmInput;
11 import com.maxeler.maxcompiler.v2.statemachine.DFEsmOutput;
12 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateEnum;
13 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateValue;
14 import com.maxeler.maxcompiler.v2.statemachine.kernel.KernelStateMachine;
15 import com.maxeler.maxcompiler.v2.statemachine.types.DFEsmValueType;
16
17 class SimpleSMStateMachine extends KernelStateMachine {
18     enum Modes {
19         COUNTING_UP, HOLD_COUNT, COUNTING_DOWN
20     }
21
22     // I/O
23     private final DFEsmOutput oCount;
24     private final DFEsmInput iMax;
25
26     private final int mHoldCount;
27
28     // State
29     private final DFEsmStateValue sCounter;
30     private final DFEsmStateValue sHoldCounter;
31     private final DFEsmStateEnum<Modes> sMode;
32
33     public SimpleSMStateMachine(KernelLib owner, int width, int holdCount) {
34         super(owner);
35         mHoldCount = holdCount;
36         DFEsmValueType counterType = dfeUInt(width);
37         // I/O
38         oCount = io.output("count", counterType);
39         iMax = io.input("max", counterType);
40         // State
41         sMode = state.enumerated(Modes.class, Modes.COUNTING_UP);
42         sCounter = state.value(counterType, 0);
43         sHoldCounter = state.value(counterType, 0);
44     }

```

The Output Function for this example simply assigns the state variable `sCounter` to the output `oCount` every cycle:

```

85 protected void outputFunction() {
86     oCount <= sCounter;
87 }

```

## 2.2 The Kernel

[Listing 3](#) shows the source code for a Kernel that integrates the simple state machine. The Kernel itself simply connects the input and output streams from the state machine directly to its own input and output streams.

*Listing 2: Program for the simple state machine (Part 2 of SimpleSMStateMachine.maxj).*

```

45  @Override
46  protected void nextState() {
47      SWITCH(sMode) {
48          CASE(Modes.COUNTING_UP) {
49              IF(sCounter == iMax) {
50                  sMode.next <= Modes.HOLD_COUNT;
51              } ELSE {
52                  /* We need to check that we are not larger than
53                   * m_max, as it might have changed
54                   */
55                  IF(sCounter > iMax) {
56                      sCounter.next <= iMax;
57                      sMode.next <= Modes.HOLD_COUNT;
58                  } ELSE {
59                      sCounter.next <= sCounter + 1;
60                  }
61              }
62          }
63          CASE(Modes.COUNTING_DOWN) {
64              IF(sCounter == 0) {
65                  sCounter.next <= sCounter + 1;
66                  sMode.next <= Modes.COUNTING_UP;
67              } ELSE {
68                  sCounter.next <= sCounter - 1;
69              }
70          }
71          OTHERWISE {
72              IF (sHoldCounter == mHoldCount)
73              {
74                  sHoldCounter.next <= 0;
75                  sMode.next <= Modes.COUNTING_DOWN;
76              }
77              ELSE {
78                  sHoldCounter.next <= sHoldCounter+1;
79              }
80          }
81      }
82  }
83
84  @Override
85  protected void outputFunction() {
86      oCount <= sCounter;
87  }
88  }

```

*Listing 3: Integrating the simple state machine into a Kernel (SimpleSMKernel.maxj).*

```

1  /**
2   * Document: MaxCompiler State Machine Tutorial (maxcompiler-sm-tutorial.pdf)
3   * Example: 1   Name: Simple statemachine
4   * MaxFile name: SimpleSM
5   * Summary:
6   *   A simple kernel that send inputs from "max" to the state machine
7   *   and sends the output of this statemachine to count.
8   */
9  package simplesm;
10
11  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
12  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
13  import com.maxeler.maxcompiler.v2.kernelcompiler.SMIO;
14  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
15
16  class SimpleSMKernel extends Kernel {
17
18      private static final int width = 32;
19
20      SimpleSMKernel(KernelParameters parameters, int holdCount) {
21          super(parameters);
22
23          DFEVar max = io.input("max", dfeUInt(32));
24
25          SMIO mySimpleSM = addStateMachine("SimpleSM", new SimpleSMStateMachine(
26              this, width, holdCount));
27
28          mySimpleSM.connectInput("max", max);
29
30          DFEVar count = mySimpleSM.getOutput("count");
31
32          io.output("count", count, count.getType());
33      }
34  }

```


### 3 State Variables

State variables can pass values from one cycle to the next. The next value for a state variable is set via the `.next` field of the variable. The next value becomes the current value the following cycle.

There are two kinds of state variables:


- Enumerated state variables: `DFEsmStateEnum`
- Value state variables: `DFEsmStateValue`

The initial value of a variable after a reset is undefined unless explicitly specified when the variable is initialized.

 New state variables cannot be defined in either the Next State Function or Output Function.

#### 3.1 Value Assignment Operator

Outputs and the next value for state variables can only be set using the value assignment operator (`<==`).

 The Java reference assignment operator cannot be used to set the `next` field on a state variable.

In Example 1, a counter is implemented by reading the *current* state of a state variable `sCounter` and assigning the result of an expression adding 1 to this to the `next` field of the same state variable:

```
59 sCounter.next <== sCounter + 1;
```

#### 3.2 Enumerated State Variables

In Example 1, an enumerated state variable (`DFEsmStateEnum`) is used to track the current mode of the state machine. The variable itself is declared as a field of the class:

```
31 private final DFEsmStateEnum<Modes> sMode;
```

The set of possible states is defined in the Java `enum` called `Modes`:

```
18 enum Modes {
19     COUNTING_UP, HOLD_COUNT, COUNTING_DOWN
20 }
```

The enumerated state variable is initialized with a reference to the `Modes` class as the first argument and the initial value as the second:

```
41 sMode = state.enumerated(Modes.class, Modes.COUNTING_UP);
```

### 3 State Variables

The next value is set using the value assignment operator:

```
50 sMode.next <== Modes.HOLD_COUNT;
```

#### 3.3 Value State Variables

Similarly to DFEVars in a Kernel, you can do operations like add, subtract, shift etc. on a value state variable (DFEsmStateValue) using overloaded operators. The type of a value state variable (DFEsmValueType) is similar to DFEType in a Kernel. State value variables have a signed or unsigned integer type (dfeInt(), dfeUInt()). dfeBool() is the same as dfeUInt(1). These types are identical to the integer types (dfeInt(), dfeUInt()) in a Kernel.

 There are no fixed-point or floating-point types available in a state machine.

In Example 1, a value state variable is used for the counter. The variable itself is declared as a field of the class:

```
29 private final DFEsmStateValue sCounter;
```

The value state variable is initialized with an integer type as the first argument and the initial value as the second:

```
42 sCounter = state.value(counterType, 0);
```

The counter is updated using the value assignment operator:

```
59 sCounter.next <== sCounter + 1;
```

## 4 Sequential Statements

The Next State Function and Output Functions are made up of three basic types of sequential statements that define the behavior of the state machine:

- Value assignments (`<==`)
- IF statements
- SWITCH statements

### 4.1 The Next State Function

Within the Next State Function, the next value of a state variable can be set using the value assignment operator (`<==`) to the `next` field. The next value for a state variable can be set multiple times in the Next State Function: the last value set within the function in a cycle is the value that the variable will take. If no value assignment is made in a cycle, then the variable holds its previous value to the next cycle.



You cannot assign to an output in the Next State Function.

Example 1 demonstrates use of the value assignment operator for the next state:

```

65     sCounter.next <== sCounter + 1;
66     sMode.next <== Modes.COUNTING_UP;

```

### 4.2 The Output Function

Within the Output Function, outputs can be set using the value assignment operator (`<==`). An output can be set multiple times in the Output Function: the last value set within the function in a cycle is the value that the output will take. The Output Function can read directly from inputs and state variables and assign to outputs.



If an output is not written to in a cycle, then its value will be undefined for that cycle.

Example 1 demonstrates use of the value assignment operator for the output:

```

85     protected void outputFunction() {
86         oCount <== sCounter;
87     }

```

Assigning an expression using an input to an output creates a combinational path from the input to the output. To insert a register between an input and an output, use the Next State Function to assign the input to a state variable, then read from this state variable in the Output Function.



You cannot assign to the `next` field of a state variable in the Output Function.

### 4.3 Conditional Statements

To conditionally execute different value assignments to state variables or outputs within the state machine, MaxJ introduces IF statements and SWITCH statements. These have similar syntax to the Java `if` and `switch` statements, but determine the behavior of the state-machine.

#### 4.3.1 IF statements

The condition in an IF statement must be an expression based on state or inputs. IF and ELSE statements can be nested in the same way as Java `if` statements. Example 1 shows the IF statement in use in the `NextState()` function:

```
49         IF(sCounter == iMax) {
50             sMode.next <== Modes.HOLD_COUNT;
51         } ELSE {
52             /* We need to check that we are not larger than
53              * m_max, as it might have changed
54              */
55             IF(sCounter > iMax) {
56                 sCounter.next <== iMax;
57                 sMode.next <== Modes.HOLD_COUNT;
58             } ELSE {
59                 sCounter.next <== sCounter + 1;
60             }
61         }
```

#### 4.3.2 SWITCH statements

A common design pattern is the use of a SWITCH statement to execute the appropriate set of actions for a current enumerated state value.

The condition in a SWITCH statement must be an expression based on state or inputs. Each case of a SWITCH statement is defined using a CASE statement where the value specified is a Java `enum`, `long` or `BigInteger`. The default case is defined using the OTHERWISE statement.



Unlike a Java `switch` statement, there is no fall-through between cases (and therefore no equivalent to the `break` statement).



Example 1 uses a SWITCH statement on an enumerated state variable in the Next State Function to determine the current mode and execute the appropriate action:

```

47  SWITCH(sMode) {
48      CASE(Modes.COUNTING_UP) {
49          IF(sCounter == iMax) {
50              sMode.next <= Modes.HOLD_COUNT;
51          } ELSE {
52              /* We need to check that we are not larger than
53               * m_max, as it might have changed
54               */
55              IF(sCounter > iMax) {
56                  sCounter.next <= iMax;
57                  sMode.next <= Modes.HOLD_COUNT;
58              } ELSE {
59                  sCounter.next <= sCounter + 1;
60              }
61          }
62      }
63      CASE(Modes.COUNTING_DOWN) {
64          IF(sCounter == 0) {
65              sCounter.next <= sCounter + 1;
66              sMode.next <= Modes.COUNTING_UP;
67          } ELSE {
68              sCounter.next <= sCounter - 1;
69          }
70      }
71      OTHERWISE {
72          IF (sHoldCounter == mHoldCount)
73          {
74              sHoldCounter.next <= 0;
75              sMode.next <= Modes.COUNTING_DOWN;
76          }
77          ELSE {
78              sHoldCounter.next <= sHoldCounter+1;
79          }
80      }

```

## 5 Kernel State Machines


Kernel state machine blocks are intended to simplify the creation of complex control flow for streaming Kernels. A typical Kernel should require only one or a small number of state machine blocks. The outputs of a state machine block can be connected to the control inputs of many other elements within a Kernel, such as:

- Boolean signals:
  - Control inputs to controlled inputs and outputs
  - Select inputs to multiplexers
  - Reset inputs to stream maximum and minimum blocks
  - Enable inputs to counters
  - Write-enable to internal memories
- Integer values:
  - Address inputs to internal memories
  - Offsets for dynamic offsets

While it is technically possible to implement an entire Kernel in a state machine block, this would be much more complex than implementing the design using the streaming paradigm of a Kernel. In general, you should avoid using a state machine block to produce arithmetic data and rarely should you need to input arithmetic data from a Kernel.

### 5.1 Inputs and Outputs

A Kernel state machine can have both stream inputs and outputs and scalar inputs and outputs. All inputs are instances of `DFEsmInput` and all outputs are instances of `DFEsmOutput`: the streaming or scalar nature is determined when the input or output is set up, for example in the constructor.

 New inputs and outputs cannot be defined in either the Next State Function or the Output Function.

All stream inputs get a new value every cycle and all outputs produce a new value every cycle. In Example 1, the stream inputs and outputs are declared as fields of the class:

```

22 // I/O
23 private final DFEsmOutput oCount;
24 private final DFEsmInput iMax;
```

They are initialized as streaming inputs and outputs in the constructor:

```

38 oCount = io.output("count", counterType);
39 iMax = io.input("max", counterType);
```

## 5.2 Output Latency

The number of cycles before the first valid value appears from an output is referred to as its *latency*. Each output can have its own latency.

Latencies are specified per output when the output is initialized, for example to initialize an output `oCount` with a latency of 2 cycles:

```
m.count = io.output("count", counterType, 2);
```



The default latency of both scalar and stream outputs is 0.

Any outputs that appear before the number of cycles specified by the latency are discarded.



After the first output has been produced by the state machine, the Kernel will receive data from the state machine every cycle until it is reset: you must ensure that the output is valid data every cycle.

Whether an output needs to have a latency specified depends on the application. For example, an output that depends on a state variable that has a valid initial value may still have a latency of 0. An output that is connected directly to an input will usually require a latency of 0.



If specifying latencies on outputs, you must ensure that the state machine produces the same number of outputs as it receives inputs.

## 5.3 Input Validity

At certain points while the Kernel is running, the inputs to the state machine will be invalid. For a Kernel that has no scalar outputs and no latency specified on any stream outputs, this is not an issue.

There is a method, `io.isValid()`, that indicates whether the inputs each cycle are valid by returning `true` if they are valid and `false` if not. If a Kernel State Machine has one or more outputs with a latency specified, then `io.isValid()` should be used to ensure that invalid data does not affect the output value.

Typically, the data will become invalid at the end of the input data streams.

## 6 Manager state machines

Sometimes the streaming model is not appropriate for a component in a Manager. A Manager state machine block allows you to implement state machine logic and integrate it seamlessly into the streaming model of the Manager.

Within a custom Manager, you can include any number of Manager state machines. Manager state machines start running as soon as the DFE is configured with the `.max` file. Each Manager state machine runs independently from all other Manager blocks, Manager state machines and Kernels. Manager state machines are only reset when the device is reset.

Inputs and outputs in a Manager state machine have additional control signals and protocols to which your state machine must conform in order to transfer data successfully. Unlike Kernel state machine outputs, you cannot define a latency on the output of a Manager state machine and have Max-Compiler automatically delay the first output: you have to manage this yourself in your state machine i.e. by not attempting to output anything until you have valid data.

There are two types of input and output interface in a Manager state machine:

- Push (`DFEsmPushInput` and `DFEsmPushOutput`): The source indicates that data is ready and requires that the data be read by the sink.
- Pull (`DFEsmPullInput` and `DFEsmPullOutput`): The sink requests data from the source.

Inputs and Outputs in a Manager state machine have additional fields for their associated control signals.



You can only assign to the IO control signals in the Output Function.

### 6.1 Push IO

Push inputs and outputs are initialized with a name to identify them, the type of the data being passed and a **stall latency** (see [subsubsection 6.1.1](#)):

```
io.DFEsmPushInput pushInput(String name, DFEsmValueType type, int stallLatency)
io.DFEsmPushOutput pushOutput(String name, DFEsmValueType type, int stallLatency)
```

The additional control signals for a push input and output are shown in [Figure 4](#). The source sets the data and sets the valid signal to 1 on the same cycle and these must be read by the sink in the same cycle.



If the sink does not read the data when the valid signal is set to 1, then the data will be lost.

#### 6.1.1 Stall Latency

Both push inputs and push outputs specify a stall latency. For an input, this is the number of data inputs that it can receive after the stall signal is set to 1. For an output, this is the number of data outputs that it will produce after receiving the stall signal. The stall latency must be greater than 0.

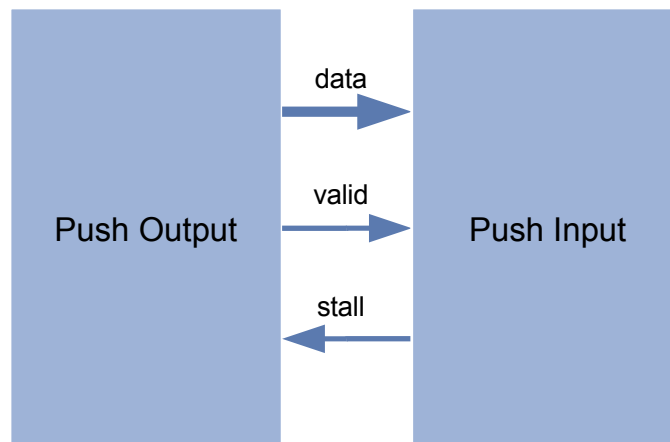


Figure 4: Data and control signals for a push interface.

For a stall latency of  $N$ , the sink must be prepared to receive  $N$  more data items when the stall signal is set to 0.

If a push input with a stall latency of  $N$  data items is connected to a push output specifying a stall latency of  $M$  data items and  $M > N$ , the Manager Compiler will insert buffering before the input to ensure that no data is dropped.

### 6.1.2 Push IO Transfer Examples

To aid in understanding the meaning of the control signals for push IO and how the timing of the signals varies with different stall latencies, this section contains two tables that show the values of the signals as data (a stream of data labeled  $d0 \rightarrow dn$ ) is transferred over a number of cycles.

[Table 1](#) shows data transfer with a stall latency of 1. With a stall latency of 1, the source can only send one more data item after the stall signal is set to 1 by the sink. The first time the sink stalls, the source sends a data item immediately; the second time, the source sends a data item a cycle later.

[Table 2](#) shows data transfer with a stall latency of 3. With a stall latency of 3, the source can send three more data items after the stall signal is set to 1 by the sink.

cycle	data	valid	stall	result
1	d0	1	0	d0 transferred
2	-	0	0	no transfer
3	d1	1	0	d1 transferred
4	d2	1	0	d2 transferred
5	d3	1	0	d3 transferred
6	d4	1	1	sink stalls - source can only send one more data item, d4 transferred
7	-	0	1	sink still stalled, source stops sending
8	d5	1	0	sink no longer stalled - source can start sending data, d5 transferred
9	-	0	1	sink stall again, no transfer
10	d6	1	1	sink still stalled, d6 transferred
11	d7	1	0	sink no longer stalled, d7 transferred

*Table 1:* Data and control signals for a push interface with stall latency of 1.

cycle	data	valid	stall	result
1	d0	1	0	d0 transferred
2	-	0	0	no transfer
3	d1	1	0	d1 transferred
4	d2	1	0	d2 transferred
5	d3	1	0	d3 transferred
6	d4	1	1	sink stalls - source can send 3 more data items, d4 transferred
7	d5	1	1	sink still stalled - source can send 2 more data items, d5 transferred
8	d6	1	1	sink still stalled - source can send 1 more data items, d6 transferred
9	-	0	1	sink still stalled - cannot accept any more data
10	-	0	1	sink still stalled - cannot accept any more data
11	-	0	1	sink still stalled - cannot accept any more data
12	d7	1	0	sink no longer stalled - source can start sending data immediately, d7 transferred

*Table 2:* Data and control signals for a push interface with stall latency of 3.

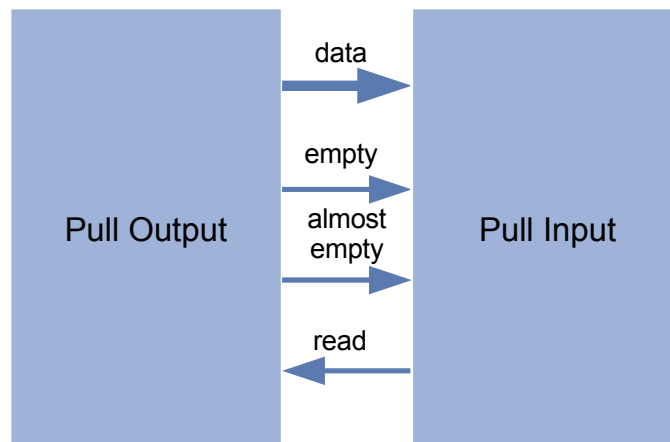


Figure 5: Data and control signals for a pull interface.

## 6.2 Pull IO

Pull inputs and outputs are initialized with a name to identify them, the type of the data being passed and, optionally, an **almost-empty threshold** (see [subsubsection 6.2.1](#)):

```
io.DFEsmPullInput pullInput(String name, DFEsmValueType type)
io.DFEsmPullInput pullInput(String name, DFEsmValueType type, int almostEmptyThreshold)
io.DFEsmPullOutput pullOutput(String name, DFEsmValueType type)
io.DFEsmPullOutput pullOutput(String name, DFEsmValueType type, int almostEmptyThreshold)
```

The additional control signals for a pull input and output are shown in [Figure 5](#).

The sink sets the read signal to 1 and the source responds *the following cycle* by setting the data. If the source sets the empty signal to 1, the sink must not request data (i.e. must set the read signal to 0) in the *same* cycle.

### 6.2.1 Almost-Empty Signal

The source can set the almost-empty signal to 1 to indicate that it is almost out of data. The **almost-empty threshold** on an output defines the number of data items remaining at the source, below which the source will set the almost-empty signal.

The almost-empty threshold on an input defines the minimum number of remaining data items at the source that the sink requires when the signal is set. This can be used, for example, if the sink reads in bursts of a number of data items and therefore requires that at least the burst-size of data is available when it starts reading.



The default almost-empty threshold is 1.



In simulation, the almost-empty threshold on a Manager state machine output is ignored and almost-empty on an input is always 1.

cycle	data	empty	almost empty	read	result
1	-	0	0	1	no transfer, read requested
2	d0	0	0	0	d0 transferred
3	-	0	0	1	no transfer, read requested
4	d1	0	0	1	d1 transferred, read requested
5	d2	0	1	1	d2 transferred, source indicates that it is almost empty (has one item left)
6	d3	1	1	0	d3 transferred, source indicates that it is empty
7	-	0	1	1	no transfer, source indicates that it is no longer empty, read requested
8	d4	0	0	1	d4 transferred, read requested
9	d5	0	0	0	d5 transferred

*Table 3:* Data and control signals for a pull interface with the default almost-empty threshold of 1.

### 6.2.2 Pull IO Transfer Examples

To aid in understanding the meaning of the control signals for pull IO and how the timing of the signals varies with different almost-empty thresholds, this section contains a number of tables that show the values of the signals as data is transferred over a number of cycles.

[Table 3](#) shows data transfer with the default almost-empty threshold of 1. [Table 4](#) shows data transfer with an almost-empty threshold of 3.



cycle	data	empty	almost empty	read	result
1	-	0	0	1	no transfer, read requested
2	d0	0	0	0	d0 transferred
3	-	0	0	1	no transfer, read requested
4	d1	0	0	1	d1 transferred, read requested
5	d2	0	1	1	d2 transferred, read requested, source indicates that it is almost empty (has three items left)
6	d3	0	1	0	d3 transferred
7	-	0	1	1	no transfer, read requested
8	d4	0	1	1	d4 transferred, read requested
9	d5	1	1	0	d5 transferred, source indicates that it is empty
10	-	0	1	1	no transfer, read requested
11	d6	0	1	0	d6 transferred
12	-	0	1	0	no transfer
13	-	0	0	1	source indicates that it has more than three items left, read requested

*Table 4:* Data and control signals for a pull interface with an almost-empty threshold of 3.

### 6.3 Pull Input to Push Output Example: Morse Code Tokenizer

The Manager state machine in Example 2 demonstrates the use of a pull input and a push output, a common combination of IO in a Manager state machine. The input is a stream of 1-bit values that carries a message in Morse code. The input is translated (tokenized) in the state machine to ASCII values of dashes, dots and spaces, which are written to the output stream.

The 1-bit stream is interpreted by the following rules:

- A dot ‘.’ is represented by a single one (1).
- A dash ‘-’ is represented by a sequence of three ones (111).
- A gap between the dots and dashes within a character is represented by a single zero (0).
- A short gap between letters is represented by a sequence of three zeroes (000).
- A gap between words is represented by a string of 7 zeroes (0000000).

For example, given the following stream of bits arriving at its input:

```
10101010001000101110101000101110101000111011101110000000000
1011101110001110111011100010111010001011101010001110101000
```

The tokenizer will recover the following sequence of dots and dashes:

```
.... . .-... .-... --- .-- --- .- .-... -..
```

This example has been written for simplicity, so the input and output of this state machine are 64-bit words, which is extremely sub-optimal for the 1-bit nature of the input and 8-bit natures of the output, but avoids adding extra code to align the data to the 128-bit (64-bit in the case of MAX2) boundaries of a CPU link.

The source code for this state machine is lengthy, so is split between the class declaration, fields and constructor in [Listing 4](#), the Next State Function in [Listing 5](#) and the Output Function in [Listing 6](#).

#### 6.3.1 Setting up the Input and Output

The state machine has a pull input called `iInput`:

```
26 private final DFEsmPullInput iInput;
```

The input is set up in the constructor:

```
53 iInput = io.pullInput("input", dfeUInt(64));
```

The state machine has a push output called `oOutput`:

```
27 private final DFEsmPushOutput oOutput;
```

The output is set up in the constructor with a stall latency of 1:

```
54 oOutput = io.pushOutput("output", dfeUInt(64), 1);
```

*Listing 4: Morse tokenizer state machine part 1: class declaration and constructor (MorseTokenizerSM-StateMachine.maxj).*

```

11 package morsetokenizersm;
12
13 import com.maxeler.maxcompiler.v2.managers.DFEManager;
14 import com.maxeler.maxcompiler.v2.statemachine.DFESmStateEnum;
15 import com.maxeler.maxcompiler.v2.statemachine.DFESmStateValue;
16 import com.maxeler.maxcompiler.v2.statemachine.manager.DFESmPullInput;
17 import com.maxeler.maxcompiler.v2.statemachine.manager.DFESmPushOutput;
18 import com.maxeler.maxcompiler.v2.statemachine.manager.ManagerStateMachine;
19
20 final class MorseTokenizerSMStateMachine extends ManagerStateMachine {
21     // types
22     private enum Mode {Start, ReadingZeros, ReadingOnes}
23     private enum Token {Pending, Invalid, Dot, Dash, DoneLetter, NewWord}
24
25     // inputs/outputs
26     private final DFESmPullInput iInput;
27     private final DFESmPushOutput oOutput;
28
29     // state
30     private final DFESmStateValue sReadDataReady;
31     private final DFESmStateValue sZeroOrOneCounter;
32     private final DFESmStateEnum<Mode> sMode;
33
34     private final DFESmStateEnum<Token> sToken;
35     private final DFESmStateValue sOutputData;
36     private final DFESmStateValue sOutputValid;
37
38     // constructor
39     MorseTokenizerSMStateMachine(DFEManager owner) {
40         super(owner);
41
42         // Initialize state
43         sZeroOrOneCounter = state.value(dfeUInt(8), 0);
44         sMode = state.enumerated(Mode.class, Mode.Start);
45         sToken = state.enumerated(Token.class, Token.Pending);
46
47         sReadDataReady = state.value(dfeBool(), false);
48
49         sOutputValid = state.value(dfeBool(), false);
50         sOutputData = state.value(dfeUInt(8));
51
52         // Set up inputs/outputs
53         iInput = io.pullInput("input", dfeUInt(64));
54         oOutput = io.pushOutput("output", dfeUInt(64), 1);
55     }

```

Listing 5: Morse tokenizer state machine part 2: Next State Function (MorseTokenizerSMStateMachine.maxj).

```

57  @Override
58  protected void nextState() {
59      // By default, we don't have a token to output
60      sToken.next <== Token.Pending;
61      /*
62       * We only get data from the input if :
63       * - the input was not empty in the previous cycle
64       * AND
65       * - the output was not stalled in the previous cycle
66       */
67      sReadDataReady.next <== !iInput.empty & !oOutput.stall;
68      /*
69       * If we get data from the input, then we can count 0s
70       * or 1s and decode the data stream */
71      IF (sReadDataReady == true) {
72          SWITCH(sMode) {
73              CASE (Mode.Start) {
74                  sZeroOrOneCounter.next <== 1;
75                  IF (iInput == 1)
76                      sMode.next <== Mode.ReadingOnes;
77                  ELSE
78                      sMode.next <== Mode.ReadingZeros;
79              }
80              CASE (Mode.ReadingOnes) {
81                  IF (iInput == 1)
82                      sZeroOrOneCounter.next <== sZeroOrOneCounter + 1;
83                  ELSE {
84                      SWITCH(sZeroOrOneCounter) {
85                          CASE (1) {sToken.next <== Token.Dot;}
86                          CASE (3) {sToken.next <== Token.Dash;}
87                          OTHERWISE {
88                              sToken.next <== Token.Invalid;
89                          }
90                      }
91                      sZeroOrOneCounter.next <== 1;
92                      sMode.next <== Mode.ReadingZeros;
93                  }
94              }
95              CASE (Mode.ReadingZeros) {
96                  IF (iInput == 1) {
97                      sZeroOrOneCounter.next <== 1;
98                      sMode.next <== Mode.ReadingOnes;
99                      SWITCH(sZeroOrOneCounter) {
100                          CASE (1) {
101                              // do nothing here, it's just the 0 between dots/dashes
102                          }
103                          CASE (2) {sToken.next <== Token.Invalid;}
104                          CASE (3) {sToken.next <== Token.DoneLetter;}
105                          CASE (4) {sToken.next <== Token.Invalid;}
106                          OTHERWISE {
107                              // we interpret anything > 4 0s as NewWord.
108                              sToken.next <== Token.NewWord;
109                          }
110                      }
111                  } ELSE {
112                      sZeroOrOneCounter.next <== sZeroOrOneCounter + 1;
113                      IF(sZeroOrOneCounter == 3) {
114                          /* Only a Token.NewWord can follow, so end the
115                           * current letter. */
116                          sToken.next <== Token.DoneLetter;
117                      }
118                  }
119              }
120          }
121      }

```

*Listing 6:* Morse tokenizer state machine part 3: Output Function (MorseTokenizerSMStateMachine.maxj).

```

123  /*
124   * Depending on the current token, set the output
125   * for the following cycle.
126   */
127  SWITCH(sToken) {
128      CASE (Token.Pending) {
129          // Don't output anything
130          sOutputValid.next <== false;
131      }
132      CASE (Token.Dot) {
133          sOutputValid.next <== true;
134          sOutputData.next <== '.';
135      }
136      CASE (Token.Dash) {
137          sOutputValid.next <== true;
138          sOutputData.next <== '-';
139      }
140      CASE (Token.DoneLetter) {
141          sOutputValid.next <== true;
142          sOutputData.next <== ' ';
143      }
144      CASE (Token.NewWord) {
145          sOutputValid.next <== true;
146          sOutputData.next <== ' ';
147      }
148      CASE (Token.Invalid) {
149          sOutputValid.next <== true;
150          sOutputData.next <== '?';
151      }
152      OTHERWISE {
153          sOutputValid.next <== false;
154      }
155  }
156 }
157
158 @Override
159 protected void outputFunction() {
160     /*
161     * We only request data from the input if :
162     * - the input is not empty
163     * AND
164     * - the output is not stalled
165     */
166     iInput.read <== !iInput.empty & !oOutput.stall;
167
168     // Output the Morse code tokens, if they are valid
169     oOutput.valid <== sOutputValid;
170     oOutput <== sOutputData.cast(oOutput.getType());
171 }

```

### 6.3.2 Read Control

The code controlling the reads from the input in this example depends on the status of the output: if the output is stalled, the state machine stops processing and does not read any more data until the output is no longer stalled.

There is code in both the Next State and Output Functions for controlling reading from the input. The read itself is requested in the Output Function, every cycle, unless the input is empty or the output is stalled:

```
166     iInput.read <== ~iInput.empty & ~oOutput.stall;
```

Every cycle, the Next State Function updates a state variables, `sReadDataReady`, to store whether a read was requested using the same expression as in the Output Function:

```
67     sReadDataReady.next <== ~iInput.empty & ~oOutput.stall;
```

This is initialized to false in the constructor to prevent erroneous data being read or written on startup:

```
47     sReadDataReady = state.value(dfeBool(), false);
```

If a read was requested in the previous cycle then data will be ready this cycle for the Next State Function to read. The first SWITCH statement in the Next State Function is then executed, in which each CASE statement reads data from the input, for example in the first CASE statement:

```
71     IF (sReadDataReady == true) {
72         SWITCH(sMode) {
73             CASE (Mode.Start) {
74                 sZeroOrOneCounter.next <== 1;
75                 IF (iInput == 1)
76                     sMode.next <== Mode.ReadingOnes;
77             ELSE
78                 sMode.next <== Mode.ReadingZeros;
79         }
```

### 6.3.3 Write Control

A write to the output is triggered whenever a token is extracted from the input stream.

The state machine has an enumerated state variable called `sToken`, which is the current token to be output. The default value of `sToken` each cycle is `Pending`, which indicates that there is no new token to output.

`oOutput` and `oOutput.valid` are set via two state variables, `sOutputData` and `sOutputValid`, in the Output Function:

```
169     oOutput.valid <== sOutputValid;
170     oOutput <== sOutputData.cast(oOutput.getType());
```

`sOutputValid` is initialized to false in the constructor to avoid writing any invalid data on startup (the data is not initialized as it will not be written out unless it is valid):

```
49     sOutputValid = state.value(dfeBool(), false);
50     sOutputData = state.value(dfeUInt(8));
```

Another SWITCH statement on `sToken` is executed every cycle. If `sToken` is Pending, then the output is set as invalid. Each of the other CASE statement outputs the appropriate ASCII value and sets the data as valid:

```

123      /*
124      * Depending on the current token, set the output
125      * for the following cycle.
126      */
127      SWITCH(sToken) {
128          CASE (Token.Pending) {
129              // Don't output anything
130              sOutputValid.next <== false;
131          }
132          CASE (Token.Dot) {
133              sOutputValid.next <== true;
134              sOutputData.next <== '.';
135          }
136          CASE (Token.Dash) {
137              sOutputValid.next <== true;
138              sOutputData.next <== '-';
139          }

```

### 6.3.4 Integrating with a Custom Manager

The Manager for Example 5 demonstrates the integration of a state machine block into a custom Manager. The source code for this Manager is shown in [Listing 7](#).

A state machine is integrated into a custom Manager by instantiating an instance of the state machine class and passing it to the `addStateMachine` method, which assigns a name to the state machine instance:

```

26      ManagerStateMachine stateMachine = new MorseTokenizerSMStateMachine(this);
27      StateMachineBlock sm = addStateMachine("MorseTokenizerSM", stateMachine);

```

The inputs and outputs to the state machine block are then connected using `getInput` and `getOutput` in the same way as other Manager blocks:

```

29
30      sm.getInput("input") <== addStreamFromCPU("input");
31
32
33      addStreamToCPU("output") <== sm.getOutput("output");

```

## 6.4 Push Input to Pull Output Example: Sorter

The Manager state machine in Example 3 demonstrates the use of a push input and a pull output. The state machine reads a number of inputs specified by `bufferSize` (set to 16 in this example), one input per cycle, and sorts them into an array of `bufferSize` value state variables as they arrive. Once the inputs are read, the state machine switches into a second mode, where it outputs the contents of the sorted array.

The source code for this state machine is split into three parts: the class declaration and constructor ([Listing 8](#)), the first part of the Next State Function ([Listing 9](#)) and the second part of the Next State Function and Output Function ([Listing 10](#)).

### 6.4.1 Setting up the Input and Output

The state machine has a push input called `iInput`:

```
25 private final DFEsmPushInput iInput;
```

The input is set up in the constructor with a stall latency of 1, which means that the input must stop sending data one cycle after the state machine sets `iInput.stall` to 1:

```
50 iInput = io.pushInput("input", dfcUInt(64), 1);
```



Listing 7: Custom Manager integrating the Manager state machine (MorseTokenizerSMManager.maxj).

```

1  /**
2   * Document: MaxCompiler State Machine Tutorial (maxcompiler-sm-tutorial.pdf)
3   * Example: 2   Name: Morse code Tokenizer
4   * MaxFile name: MorseTokenizerSM
5   * Summary:
6   *   Custom Design that creates a new MorseTokenizerSM state machine, adds
7   *   it , and wires up the inputs/outputs. All IO is between the CPU and the DFE.
8   */
9   package morsetokenizersm;
10
11  import com.maxeler.maxcompiler.v2.build.EngineParameters;
12  import com.maxeler.maxcompiler.v2.managers.BuildConfig;
13  import com.maxeler.maxcompiler.v2.managers.BuildConfig.Level;
14  import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
15  import com.maxeler.maxcompiler.v2.managers.custom.blocks.StateMachineBlock;
16  import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
17  import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
18  import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
19  import com.maxeler.maxcompiler.v2.statemachine.manager.ManagerStateMachine;
20
21  class MorseTokenizerSMManager extends CustomManager {
22
23      MorseTokenizerSMManager(EngineParameters params) {
24          super(params);
25
26          ManagerStateMachine stateMachine = new MorseTokenizerSMStateMachine(this);
27          StateMachineBlock sm = addStateMachine("MorseTokenizerSM", stateMachine);
28
29
30          sm.getInput("input") <== addStreamFromCPU("input");
31
32
33          addStreamToCPU("output") <== sm.getOutput("output");
34      }
35
36      private static EngineInterface interfaceDefault () {
37          EngineInterface ei = new EngineInterface();
38
39          InterfaceParam maxInputSize = ei.addParam("maxInputSize", CPUTypes.INT);
40          InterfaceParam resultSize = ei.addParam("resultSize", CPUTypes.INT);
41
42          ei.setStream("input", CPUTypes.UINT64, CPUTypes.UINT64.sizeInBytes() * maxInputSize);
43          ei.setStream("output", CPUTypes.UINT64, CPUTypes.UINT64.sizeInBytes() * resultSize);
44
45          return ei;
46      }
47
48      public static void main(String args[]) {
49          MorseTokenizerSMManager m = new MorseTokenizerSMManager(new EngineParameters(args));
50          m.setBuildConfig(new BuildConfig(Level.FULL_BUILD));
51          m.createSLiCInterface(interfaceDefault());
52          m.build();
53      }
54  }
55

```

Listing 8: Sorter state machine part 1: class declaration and constructor (SorterSMStateMachine.maxj).

```

8  package sortersm;
9
10 import com.maxeler.maxcompiler.v2.managers.DFEManager;
11 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateEnum;
12 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateValue;
13 import com.maxeler.maxcompiler.v2.statemachine.manager.DFEsmPullOutput;
14 import com.maxeler.maxcompiler.v2.statemachine.manager.DFEsmPushInput;
15 import com.maxeler.maxcompiler.v2.statemachine.manager.ManagerStateMachine;
16 import com.maxeler.maxcompiler.v2.utils.MathUtils;
17
18 public final class SorterSMStateMachine extends ManagerStateMachine {
19     // Size of the sorting buffer
20     private final int mBufferSize = 16;
21     // types
22     private enum Mode {Sorting, Outputting}
23
24     // inputs/outputs
25     private final DFEsmPushInput iInput;
26     private final DFEsmPullOutput oOutput;
27
28     // state
29     private final DFEsmStateEnum<Mode> sMode;
30     private final DFEsmStateValue[] sSortedBuffer;
31
32     private final DFEsmStateValue sInputCounter;
33     private final DFEsmStateValue sOutputCounter;
34
35     private final DFEsmStateValue sInputData;
36     private final DFEsmStateValue sInputValid;
37
38     private final DFEsmStateValue sOutputData;
39     private final DFEsmStateValue sOutputEmpty;
40     private final DFEsmStateValue sOutputAlmostEmpty;
41
42     // For debugging
43     // private final DFEsmStateValue sCycleCounter = state.value(dfeUInt(64), 0);
44
45     // constructor
46     public SorterSMStateMachine(DFEManager owner) {
47         super(owner);
48
49         // Set up inputs/outputs
50         iInput = io.pushInput("input", dfeUInt(64), 1);
51         oOutput = io.pullOutput("output", dfeUInt(64), mBufferSize);
52
53         // Initialize state
54         sSortedBuffer = new DFEsmStateValue[mBufferSize];
55         for (int i = 0; i < mBufferSize; i++)
56             sSortedBuffer[i] = state.value(dfeUInt(64), 0);
57
58         sMode = state.enumerated(Mode.class, Mode.Sorting);
59
60         sInputData = state.value(dfeUInt(64));
61         sInputValid = state.value(dfeBool(), false);
62         sInputCounter = state.value(dfeUInt(MathUtils.bitsToAddress(mBufferSize)), 0);
63
64         sOutputData = state.value(dfeUInt(64));
65         sOutputEmpty = state.value(dfeBool(), true);
66         sOutputAlmostEmpty = state.value(dfeBool(), true);
67         sOutputCounter = state.value(dfeUInt(MathUtils.bitsToAddress(mBufferSize)), 0);
68     }

```

*Listing 9: Sorter state machine part 2: First half of Next State Function(SorterSMStateMachine.maxj).*

```

70  @Override
71  protected void nextState() {
72      sInputValid.next <== iInput.valid;
73      sInputData.next <== iInput;
74      SWITCH (sMode)
75      {
76          CASE (Mode.Sorting)
77          {
78              IF (sInputValid)
79              {
80                  IF (sInputCounter <= mBufferSize-1)
81                      sInputCounter.next <== sInputCounter+1;
82                  ELSE
83                      sInputCounter.next <== 0;
84
85                  IF (sInputCounter == (mBufferSize-1))
86                  {
87                      sMode.next <== Mode.Outputting;
88                      sOutputAlmostEmpty.next <== false;
89                      sOutputEmpty.next <== false;
90                  }
91
92                  IF (sInputCounter == 0)
93                      sSortedBuffer[mBufferSize-1].next <== sInputData;
94
95                  for (int i = 0; i < mBufferSize; i++)
96                  {
97                      IF (sInputData >= sSortedBuffer[i])
98                      {
99                          if (i!=0)
100                              sSortedBuffer[i-1].next <== sSortedBuffer[i];
101
102                          if (i!=mBufferSize-1)
103                          {
104                              IF (sInputData < sSortedBuffer[i+1])
105                              {
106                                  sSortedBuffer[i].next <== sInputData;
107                              }
108                          }
109                          else
110                              sSortedBuffer[i].next <== sInputData;
111                      }
112                  }
113              }
114          }

```

*Listing 10: Sorter state machine part 3: Second half of Next State Function and Output Function (SorterSMStateMachine.maxj).*

```

115     CASE(Mode.Outputting)
116     {
117         IF (oOutput.read === true)
118         {
119             sOutputData.next <= sSortedBuffer[mBufferSize-1];
120
121             for (int i = 0; i < mBufferSize-1; i++)
122             {
123                 sSortedBuffer[i+1].next <= sSortedBuffer[i];
124             }
125             sSortedBuffer[0].next <= 0;
126
127             IF (sOutputCounter <= mBufferSize-1)
128                 sOutputCounter.next <= sOutputCounter+1;
129             ELSE
130                 sOutputCounter.next <= 0;
131
132             IF (sOutputCounter === 0)
133                 sOutputAlmostEmpty.next <= true;
134
135             IF (sOutputCounter === (mBufferSize - 1))
136             {
137                 sMode.next <= Mode.Sorting;
138                 sOutputEmpty.next <= true;
139             }
140         }
141     }
142 }
143 /*
144  * Debug code - uncomment to see cycle-by-cycle updates of the sorted buffer
145  * - also uncomment the declaration of sCycleCounter above
146  */
147 /*
148 sCycleCounter.next <= sCycleCounter + 1;
149 debug.simPrintf("Cycle: %d, m_mode: %s, m_inputData: %d, m_inputValid: %d, m_inputCounter: %d\n",
150     sCycleCounter, sMode, sInputData, sInputValid, sInputCounter);
151 debug.simPrintf(" sortedBuffer: ");
152 for (int i = 0; i < mBufferSize; i++)
153 {
154     debug.simPrintf("%d ", sSortedBuffer[i]);
155 }
156 debug.simPrintf("\n");
157 */
158 }
159
160 @Override
161 protected void outputFunction() {
162     ilInput.stall <=
163         // Don't accept inputs while outputting data
164         (sMode === Mode.Outputting)
165         // Stop accepting inputs when we have finished sorting
166         | (sMode === Mode.Sorting
167         /* When there is only one more free place in the buffer and we
168          * are reading input data in the current cycle, assert stall
169          * (as the input we're reading will take the last free position
170          * in the buffer in the next cycle). This code depends on an
171          * input latency of 0.*/
172         & ((sInputCounter === (mBufferSize - 2)) & ilInput.valid)
173         // Assert stall when the buffer is full.
174         | (sInputCounter === (mBufferSize - 1))
175         );
176     oOutput <= sOutputData;
177     oOutput.empty <= sOutputEmpty;
178     oOutput.almostEmpty <= sOutputAlmostEmpty;
179 }

```

## 6 Manager state machines

The state machine has a pull output called `oOutput`:

```
26 private final DFEsmPullOutput oOutput;
```

The output is set up in the constructor with an almost-empty threshold of `bufferSize` (set to 16 in this example), which means that once the state machine has set `oOutput.almostEmpty` to 1, the receiver can request no more than 16 data items:

```
51 oOutput = io.pullOutput("output", dfeUInt(64), mBufferSize);
```

### 6.4.2 Read Control

Two value state variables are used to store the status of the input:

```
35 private final DFEsmStateValue sInputData;
36 private final DFEsmStateValue sInputValid;
```

In the Output Function, the stall signal on the input, `iInput.stall`, is set depending on the state of the sorter:

```
161 protected void outputFunction() {
162     iInput.stall <==
163     // Don't accept inputs while outputting data
164     (sMode === Mode.Outputting)
165     // Stop accepting inputs when we have finished sorting
166     | (sMode === Mode.Sorting
167     /* When there is only one more free place in the buffer and we
168     * are reading input data in the current cycle, assert stall
169     * (as the input we're reading will take the last free position
170     * in the buffer in the next cycle). This code depends on an
171     * input latency of 0.*/
172     & ((sInputCounter === (mBufferSize - 2)) & iInput.valid)
173     // Assert stall when the buffer is full .
174     | (sInputCounter === (mBufferSize - 1))
175     );
```

The stall signal is set to 1 when `sInputCounter` is equal to `bufferSize-2` because it takes a cycle for the data from `iInput` to propagate to `sInputData`.

In the Next State Function, `iInput` and `iInput.valid` are read into value state variables every cycle:

```
71 protected void nextState() {
72     sInputValid.next <== iInput.valid;
73     sInputData.next <== iInput;
```

Importantly, the state variable `sInputValid` is initialized to `false` in the constructor to ensure that no erroneous data is read when the state machine starts (the data does not need to be initialized as `sInputValid` will be `false`):

```
60 sInputData = state.value(dfeUInt(64));
61 sInputValid = state.value(dfeBool(), false);
```

This arrangement means that `sInputValid` can be read in the Next State Function to receive data from the input, with a latency of one cycle.

The main SWITCH statement swaps between two modes: sorting the input and outputting the sorted data. When sorting the input, the state machine uses a value state variable `sInputCounter` to keep track of the number of inputs it has read:

```

74     SWITCH (sMode)
75     {
76         CASE (Mode.Sorting)
77         {
78             IF (sInputValid)
79             {
80                 IF (sInputCounter <= mBufferSize-1)
81                     sInputCounter.next <== sInputCounter+1;
82                 ELSE
83                     sInputCounter.next <== 0;

```

### 6.4.3 Write Control

Three value state variables are used to store the status of the output:

```

38     private final DFEsmStateValue sOutputData;
39     private final DFEsmStateValue sOutputEmpty;
40     private final DFEsmStateValue sOutputAlmostEmpty;

```

In the Output Function, these are all attached directly to the output:

```

161     protected void outputFunction() {
162         iInput . stall <==
163             // Don't accept inputs while outputting data
164             (sMode === Mode.Outputting)
165             // Stop accepting inputs when we have finished sorting
166             | (sMode === Mode.Sorting
167                 /* When there is only one more free place in the buffer and we
168                  * are reading input data in the current cycle, assert stall
169                  * (as the input we're reading will take the last free position
170                  * in the buffer in the next cycle). This code depends on an
171                  * input latency of 0.*/
172                 & ((sInputCounter === (mBufferSize - 2)) & iInput . valid)
173                 // Assert stall when the buffer is full .
174                 | (sInputCounter === (mBufferSize - 1))
175             );
176         oOutput <== sOutputData;
177         oOutput.empty <== sOutputEmpty;
178         oOutput.almostEmpty <== sOutputAlmostEmpty;
179     }

```

Importantly, the two state variables `sOutputEmpty` and `sOutputAlmostEmpty` are initialized to `true` in the constructor to ensure that no erroneous data is written when the state machine starts (the data does not need to be initialized as `sOutputEmpty` will be `true`):

```

64     sOutputData = state.value(dfeUInt(64));
65     sOutputEmpty = state.value(dfeBool(), true);
66     sOutputAlmostEmpty = state.value(dfeBool(), true);

```

## 6 Manager state machines

Empty and almost-empty are simultaneously set to 0 to allow the receiver to start requesting data the cycle *after* the sorting process is complete:

```

85         IF (sInputCounter === (mBufferSize-1))
86         {
87             sMode.next <== Mode.Outputting;
88             sOutputAlmostEmpty.next <== false;
89             sOutputEmpty.next <== false;
90         }

```

When the state machine is in the outputting mode, if a read is requested in a cycle, then the value state variable sOutputData is set, which will then output the data the *following cycle*:

```

115     CASE(Mode.Outputting)
116     {
117         IF (oOutput.read === true)
118         {
119             sOutputData.next <== sSortedBuffer[mBufferSize-1];

```

Similarly to when sorting the input, the state machine uses a value state variable sOutputCounter to keep track of the number of outputs it has written. When the state machine has started outputting data, almost-empty is set to 1, indicating that the receiver can now read only 16 outputs:

```

132         IF (sOutputCounter === 0)
133         {
134             sOutputAlmostEmpty.next <== true;

```

When the all of the data has been output, empty is then set to 1, and reads are restarted:

```

135         IF (sOutputCounter === (mBufferSize - 1))
136         {
137             sMode.next <== Mode.Sorting;
138             sOutputEmpty.next <== true;
139         }

```

### 6.5 Selecting IO Types

You can choose the input and output types that are best suited to the state machine that you are designing.

If the stall latencies between a push output and a push input don't match, then the Manager Compiler will insert extra buffering to ensure that no data is lost between the output and the input. Likewise, if the almost-empty thresholds between a pull output and a pull input do not match, the Manager Compiler will insert extra buffering to ensure that the receiver gets its required almost-empty latency. Where a push IO is connected to a pull IO, the Manager Compiler will insert an adapter and some buffering if required.

When choosing which IO type to use, consider the following guidelines:

- Is the state machine block passive or active? For example, a FIFO would more naturally have a push input and a pull output. A decoder or encoder that produces a different amount of data than it consumes would more naturally have a pull input and a push output.
- Where are any buffers required in the system? Using a pull input removes the requirement for buffering data at the input to a state machine and using a push output removes the requirement for buffering data at an output.

## 7 Scalar Inputs and Outputs

Scalar inputs and outputs are similar to scalar inputs and outputs in a Kernel: scalar inputs are used to transfer a single value from the CPU to a state machine block; scalar outputs transfer a single value from a state machine block to the CPU.

### 7.1 Accessing Scalar IO in a State Machine

Example 4 shows both a scalar input and a scalar output in use in a Kernel state machine. The full source for the state machine block is shown in [Listing 11](#) and [Listing 12](#).

The example state machine implements the same counter as Example 1, except that the number of cycles to hold the counter is now set by the scalar input `max`. A second counter `sMaxCounter` keeps track of the number of times that `count` hits the value specified by `max`, and outputs the current value of this counter via the scalar output `maxCount`.

The scalar inputs and outputs are declared as fields of type `DFEsmInput` and `DFEsmOutput`, in the same way as stream inputs and outputs:

```
30 private final DFEsmInput iHoldCount;
31 private final DFEsmOutput oMaxCount;
```

They are then initialized as scalar inputs and outputs with the required names:

```
44 iHoldCount = io.scalarInput("holdCount", counterType);
45 oMaxCount = io.scalarOutput("maxCount", counterType);
```

Scalar inputs and outputs are accessed identically to stream inputs and outputs:

```
57 IF(sCounter == iMax) {
```

```
92 protected void outputFunction() {
93     oCount <= sCounter;
94     oMaxCount <= sMaxCounter;
95 }
96 }
```



*Listing 11:* Program for a state machine demonstrating the use of scalar inputs and outputs (Part 1 of ScalarSMStateMachine.maxj).

```
12 package scalarsm;
13 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelLib;
14 import com.maxeler.maxcompiler.v2.statemachine.DFEsmInput;
15 import com.maxeler.maxcompiler.v2.statemachine.DFEsmOutput;
16 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateEnum;
17 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateValue;
18 import com.maxeler.maxcompiler.v2.statemachine.kernel.KernelStateMachine;
19 import com.maxeler.maxcompiler.v2.statemachine.types.DFEsmValueType;
20
21 class ScalarSMStateMachine extends KernelStateMachine {
22     enum Modes {
23         COUNTING_UP, HOLD_COUNT, COUNTING_DOWN
24     }
25
26     // I/O
27     private final DFEsmOutput oCount;
28     private final DFEsmInput iMax;
29
30     private final DFEsmInput iHoldCount;
31     private final DFEsmOutput oMaxCount;
32
33     // State
34     private final DFEsmStateValue sCounter;
35     private final DFEsmStateValue sHoldCounter;
36     private final DFEsmStateValue sMaxCounter;
37     private final DFEsmStateEnum<Modes> sMode;
```

*Listing 12:* Program for a state machine demonstrating the use of scalar inputs and outputs (Part 2 of ScalarSMStateMachine.maxj).

```

38 ScalarSMStateMachine(KernelLib owner, int width) {
39     super(owner);
40     DFEsmValueType counterType = dfeUInt(width);
41     // I/O
42     oCount = io.output("count", counterType);
43     iMax = io.input("max", counterType);
44     iHoldCount = io.scalarInput("holdCount", counterType);
45     oMaxCount = io.scalarOutput("maxCount", counterType);
46     // State
47     sMode = state.enumerated(Modes.class, Modes.COUNTING_UP);
48     sCounter = state.value(counterType, 0);
49     sHoldCounter = state.value(counterType, 0);
50     sMaxCounter = state.value(counterType, 0);
51 }
52
53 @Override
54 protected void nextState() {
55     SWITCH(sMode) {
56         CASE(Modes.COUNTING_UP) {
57             IF(sCounter == iMax) {
58                 sMode.next <= Modes.HOLD_COUNT;
59                 IF (io.isInputValid ())
60                     sMaxCounter.next <= sMaxCounter+1;
61             } ELSE {
62                 IF(sCounter > iMax) {
63                     sCounter.next <= iMax;
64                     sMode.next <= Modes.HOLD_COUNT;
65                     IF (io.isInputValid ())
66                         sMaxCounter.next <= sMaxCounter+1;
67                 } ELSE {
68                     sCounter.next <= sCounter + 1;
69                 }
70             }
71         }
72         CASE(Modes.COUNTING_DOWN) {
73             IF(sCounter == 0) {
74                 sCounter.next <= sCounter + 1;
75                 sMode.next <= Modes.COUNTING_UP;
76             } ELSE
77                 sCounter.next <= sCounter - 1;
78         }
79         OTHERWISE {
80             IF (sHoldCounter == iHoldCount)
81             {
82                 sHoldCounter.next <= 0;
83                 sMode.next <= Modes.COUNTING_DOWN;
84             }
85             ELSE
86                 sHoldCounter.next <= sHoldCounter+1;
87         }
88     }
89 }
90
91 @Override
92 protected void outputFunction() {
93     oCount <= sCounter;
94     oMaxCount <= sMaxCounter;
95 }
96 }

```

## 7.2 Accessing Scalar IO From the CPU

Scalar inputs and outputs are added to the SLiC API, just as for Kernel scalar inputs and outputs.

### 7.2.1 Kernel State Machines

Scalar inputs and outputs for a state machine instance in a Kernel are automatically added to the SLiC interface, as shown in this snippet from the header file for example 4:

```
void ScalarSM(
    int32_t param_N,
    uint64_t inscalar_ScalarSMKernel.ScalarSM.holdCount,
    uint64_t *outscalar_ScalarSMKernel.ScalarSM.maxCount,
    const uint32_t *instream_max,
    uint32_t *outstream_count);
```

The CPU code in Example 4 uses the Basic Static function:

```
120     uint64_t maxCount;
121     int holdCount = 4;
122     ScalarSM(size, holdCount, &maxCount, dataIn, dataOut);
```

Alternatively, the Advanced Dynamic functions can be used:

```
void max_set_uint64t(max_actions_t *actions, const char *block_name, const char *name, uint64_t v);
void max_set_double (max_actions_t *actions, const char *block_name, const char *name, double v);

void max_get_uint64t(max_actions_t *actions, const char *block_name, const char *name, uint64_t *v);
void max_get_double (max_actions_t *actions, const char *block_name, const char *name, double *v);
```

`block_name` is the name of the Kernel in the Manager, and the name of the scalar input or output takes the form `StateMachineName.ScalarName`. For example, to set the `holdCount` scalar input in Example 4 using the Advanced Dynamic SLiC interface, we would use:

```
max_set_uint64t(actions, "ScalarSMKernel", "ScalarSM.holdCount", 3);
```

In Java-driven simulation, state machine scalar inputs are set using a method specific to state machines:

```
35     m.setStateMachineScalarInput("ScalarSM", "holdCount", m_holdCount);
```

Likewise for reading the scalar output:

```
42     int maxCount = (int) m.getStateMachineScalarOutput("ScalarSM", "maxCount");
```

### 7.2.2 Manager State Machines

Scalar inputs and outputs for Manager state machines are added to the SLiC interface in the same way as Kernel state machine scalar inputs and outputs.

Access via the Advanced Dynamic functions requires the specifying the name of the state machine Manager block. For example, an instance of a state machine is declared with the name "SorterSM" in a custom Manager with the line:

```
ManagerStateMachine stateMachine = new SorterSMStateMachine(this);
StateMachineBlock sm = addStateMachine("SorterSM", stateMachine);
```

## 7 Scalar Inputs and Outputs

---

To access a scalar input called "myScalarInput" within this state machine, we would write:

```
max_set_uint64t(actions, "SorterSM", "myScalarInput", value);
```

## 8 Memories

State machine blocks can use the internal memory elements available on the device in a similar manner to Kernels. ROMs and RAMs allow storage of large amounts of data within a state machine, of the order of tens to thousands of items (tens of thousands or more items are generally stored in off-chip memory).

Memories can be either read-only (ROM) or read-write (RAM) and have one or two access ports to the same contents. An access port has an address input, a data output and, in the case of a RAM, write-enable and data inputs. Mapped memories can have their contents written to and, in the case of a RAM, read from the CPU.

All memories have the following methods:

- `getDepth`: returns the number of elements in the RAM/ROM.
- `getLatency`: returns the latency in number of cycles.
- `getAddressWidth`: returns the width of the address port(s) on the memory.
- `getNumPorts`: returns the number of ports on the memory.

### 8.1 Memory Latency

The latency of a memory indicates the number of cycles after setting the read signal to 1 that the data will be output.

The latency for a memory can be set between 1 and 3 cycles using an enum `Latency` when the memory is initialized:

```
Latency{ONE_CYCLE,TWO_CYCLES,THREE_CYCLES}
```

For example, initializing a single-port ROM `myMem` with a latency of three cycles:

```
DFesmSinglePortROM myMem = mem.rom(dfInt(8), Latency.THREE_CYCLES, contents);
```

For a dual-port memory, the latency applies to both ports.



Setting a higher latency for the RAM can help in meeting higher clock rates.

### 8.2 ROMs

A ROM can be used to store static data that is not written to by the state machine.

#### 8.2.1 Single-Port ROMs

Single-port ROMs (`DFesmSinglePortROM`) have one input address (`address`) and one data output (`dataOut`), that are fields of the class.

The contents of a ROM can be defined in one of three ways:

```
DFesmSinglePortROM mem.rom(DFesmValueType type, Latency latency, int... contents)
DFesmSinglePortROM mem.rom(DFesmValueType type, Latency latency, long... contents)
DFesmSinglePortROM mem.rom(DFesmValueType type, Latency latency, List<BigInteger> contents)
```

### 8.2.2 Dual-Port ROMs

Dual-port ROMs (`DFEsmDualPortROM`) have one input address per port (`addressA` and `addressB`) and one data output per port (`dataOutA` and `dataOutB`), that are fields of the class.

```
DFEsmDualPortROM mem.romDualPort(DFEsmValueType type, Latency latency, int... contents)
DFEsmDualPortROM mem.romDualPort(DFEsmValueType type, Latency latency, long... contents)
DFEsmDualPortROM mem.romDualPort(DFEsmValueType type, Latency latency, List<BigInteger> contents)
```

## 8.3 RAMs

RAMs have ports that you can either read from or write to each cycle.

### 8.3.1 Single-Port RAMs

A single-port RAM (`DFEsmSinglePortRAM`) has one port that can be either read from or written to each cycle. A single-port RAM is initialized with the type of data to be stored in the RAM (`type`), its size (`depth`), the mode of the port (`portMode`) and the required latency (`latency`):

```
DFEsmSinglePortRAM mem.ram(DFEsmValueType type, int depth, SinglePortRAMMode portMode, Latency latency)
```

Single-port RAMs can be in one of two modes:

```
SinglePortRAMMode{ READ_FIRST, WRITE_FIRST }
```

In `READ_FIRST` mode, a read in the same cycle as a write to the same location in a RAM will read the value currently stored in the RAM; in `WRITE_FIRST` mode, the read would read the new value being written into the RAM in the current cycle.



Note that using `WRITE_FIRST` mode can reduce the maximum clock frequency of the design as there is a zero-cycle path from the write input data to the read output data.

The RAM has address, data and write enable input signals and a data output signal:

- *Inputs:* address, dataIn, writeEnable
- *Outputs:* dataOut

### 8.3.2 Dual-Port RAMs

A dual-port RAM (`DFEsmDualPortRAM`) has two fully independent ports that can each either perform a read or a write within a cycle. A dual-port RAM is initialized with the type of data to be stored in the RAM (`type`), its size (`depth`), the mode of the two ports (`portModeA` and `portModeB`) and the required latency (`latency`):

```
DFEsmDualPortRAM mem.romDualPort(DFEsmValueType type, int depth, DualPortRAMMode portModeA, DualPortRAMMode
portModeB, Latency latency)
```

Dual-port RAMs can be in one of four modes:

```
DualPortRAMMode{ READ_ONLY, RW_READ_FIRST, RW_WRITE_FIRST, WRITE_ONLY }
```

READ\_ONLY disables the write inputs to the port; likewise WRITE\_ONLY disables the read address and data output for the port. RW\_READ\_FIRST mode and RW\_WRITE\_FIRST mode are the same as for single-port RAMs.



Note that using RW\_WRITE\_FIRST mode can reduce the maximum clock frequency of the design as there is a zero-cycle path from the write input data to the read output data.

For example, declaring a dual-port RAM called (myRAM) to hold 16 elements of 8-bit signed integer data with one read-write port and one read-only port:

```
DFEsmDualPortRAM myMem = mem.ramDualPort(dfeInt(8), 16, DualPortRAMMode.RW_READ_FIRST, DualPortRAMMode.READ_ONLY, Latency.THREE_CYCLES);
```

The port on the dual-port RAM has address, data and write enable input signals and a data output signal:

- Inputs:
  - Port A: addressA, dataInA, writeEnableA
  - Port B: addressB, dataInB, writeEnableB
- Outputs:
  - Port A: dataOutA
  - Port B: dataOutB

## 8.4 Mapped Memories

Mapped memories can be accessed directly from the CPU code without having to be wired through a Kernel. The name string for the memories takes the form StateMachineInstanceName.MemoryName for a Kernel state machine and MemoryName for a Manager state machine.

Mapped memories can be set from the CPU code using the same C API as for mapped memories in a Kernel:

```
void max_set_mem_uint64t(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, uint64_t v);
void max_set_mem_double (max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, double v);
```

The contents of the memory can be read back on the CPU one element at a time:

```
void max_get_mem_uint64t(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, uint64_t *v);
void max_get_mem_double (max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, double *v);
```

### 8.4.1 Mapped ROMs

Mapped ROMs take a string name that identifies the ROM:

```
DFEsmSinglePortMappedROM mem.romMapped(String name, DFEsmValueType type, int depth, Latency latency)
DFEsmDualPortMappedROM mem.romMappedDualPort(String name, DFEsmValueType type, int depth, Latency latency)
```

The contents of the ROM are not set at initialization: they must be set from the CPU before the Kernel or Manager starts.

### 8.4.2 Mapped RAMs

Mapped RAMs are convenient for loading initial data into the RAM before the Kernel or Manager starts, without having to code for this in the Manager or Kernel, and likewise to read the contents of a RAM from the CPU during or after computation.

Compared to non-mapped RAMs, mapped RAMs take an additional argument that is a name that identifies the RAM:

```
DFesmSinglePortMappedRAM mem.ramMapped(String name, DFesmValueType type, int depth, SinglePortRAMMode portMode,
    Latency latency)
DFesmDualPortMappedRAM mem.ramMappedDualPort(String name, DFesmValueType type, int depth, DualPortRAMMode
    portModeA, DualPortRAMMode portModeB, Latency latency)
```

## 8.5 Dual-Port Mapped RAM Example

Example 5 uses a dual-port RAM to generate a histogram of the distribution of intensity of a gray-scale image. The image is received one pixel per cycle on the input `iImage`. The range of intensities is divided up into `m_histogramSize` equally-sized buckets, which are stored in a RAM. The count for a bucket is incremented each time the intensity for an input pixel falls within its range.

### 8.5.1 The state machine

The difficulty with using a RAM to store the data is that the RAM has a latency of one or more cycles for a read, so it is not possible to simply increment an element in a single cycle. This can be worked around, however, by using a dual-port RAM: one port is used to read the current value of the RAM and the second port is used to write the new value back into the RAM.

The source code for the Kernel for this example is shown in [Listing 13](#) and [Listing 14](#).

The RAM is declared as a mapped dual-port RAM:

```
40 private final DFesmDualPortMappedRAM sHistogram;
```

This is then set up to have one read port and one write port in the constructor:

```
65 sHistogram = mem.ramMappedDualPort(
66     "histogram",
67     mDataType,
68     mHistogramSize,
69     DualPortRAMMode.READ_ONLY,
70     DualPortRAMMode.WRITE_ONLY,
71     Latency.ONE_CYCLE);
```

Every cycle, the address to the read port of the RAM is set to the bucket of the incoming pixel (using a shift for efficiency), and this address is also stored in a value state variable:

```
79 // Set off a read from the RAM for the bucket where the current input falls
80 sHistogram.addressA <== (iImage>>mShiftAmount).cast(mAddressType);
81
82 // Store the address that we started the read for
83 sAddressReg.next <== (iImage>>mShiftAmount).cast(mAddressType);
```

Every cycle, the write enable signal on the write port is set to `false` by default:

```
85 // By default, we won't write
86 sHistogram.writeEnableB <== false;
```



If the data read the previous cycle was valid, then the bucket for the incoming pixel is compared with the previous pixel. If they are the same, then the write back to RAM is disabled and a value state variable `sAddValue` is incremented. If they are not the same, then the new value is written back into the RAM:

```

88      // If the input was valid in the previous cycle
89      IF (sInputValidReg === true)
90      {
91          /*
92           * Check if the current input is different to the previous input, or
93           * if we have read all the data.
94           *
95           * If so, then we enable the write to the RAM.
96           */
97          IF (sAddressReg !== (iImage>>mShiftAmount).cast(mAddressType) | sFinished)
98          {
99              sHistogram.writeEnableB <== true;
100              sAddValue.next <== 0;
101          }
102          // Otherwise, we hold the write to the RAM and increment the amount we will add
103          ELSE
104          {
105              sHistogram.writeEnableB <== false;
106              sAddValue.next <== sAddValue+1;
107          }
108      }

```

The input data and address are always written to the RAM as this will have no effect without the write enable signal being true:

```

110      // Always write out the data and address, as this has no effect without the write enable
111      sHistogram.dataInB <== sHistogram.dataOutA+sAddValue+1;
112      sHistogram.addressB <== sAddressReg;

```

Finally, the amount of data that has been read is kept track of and a Boolean value state variable `sFinished` is set to true when the last item is read: this is used to force the write of the last pixel to the RAM as there will not be an incoming pixel to compare against:

```

110      // Always write out the data and address, as this has no effect without the write enable
111      sHistogram.dataInB <== sHistogram.dataOutA+sAddValue+1;
112      sHistogram.addressB <== sAddressReg;

```

### 8.5.2 The CPU code

The CPU code (shown in [Listing 15](#)) uses the Basic Static SLiC interface to run the DFE. The initial mapped memory contents are set to 0 and space allocated for the final contents read back from the DFE:

```

92      uint64_t *ramIn = malloc(32 * sizeof(uint64_t));
93      uint64_t* outHistogram = malloc(histogramSize * sizeof(uint64_t));
94      for (int i = 0; i < histogramSize; i++)
95          ramIn[i] = 0;

```

These are then passed to the Basic Static function, so that the mapped memory is set up before

the state machine starts and then read back when the state machine completes:

```
101 HistogramSM(width * height, inImage, outImage, ramIn, outHistogram);
```

*Listing 13:* Histogram state machine part 1: class declaration and constructor (HistogramSMStateMachine.maxj).

```

9  package histogramsm;
10
11  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelLib;
12  import com.maxeler.maxcompiler.v2.statemachine.Latency;
13  import com.maxeler.maxcompiler.v2.statemachine.DFEsmDualPortMappedRAM;
14  import com.maxeler.maxcompiler.v2.statemachine.DFEsmInput;
15  import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateValue;
16  import com.maxeler.maxcompiler.v2.statemachine.kernel.KernelStateMachine;
17  import com.maxeler.maxcompiler.v2.statemachine.stdlib.Mem.DualPortRAMMode;
18  import com.maxeler.maxcompiler.v2.statemachine.types.DFEsmValueType;
19  import com.maxeler.maxcompiler.v2.utils.MathUtils;
20
21  class HistogramSMStateMachine extends KernelStateMachine {
22      private final int mHistogramSize;
23      private final int mShiftAmount;
24
25      // Types
26      private final DFEsmValueType mAddressType;
27      private final DFEsmValueType mDataType;
28
29      // I/O
30      private final DFEsmInput ilImage;
31      private final DFEsmInput ilImageSize;
32
33      // State
34      private final DFEsmStateValue sAddressReg;
35      private final DFEsmStateValue sInputValidReg;
36      private final DFEsmStateValue sAddValue;
37      private final DFEsmStateValue sInputCounter;
38      private final DFEsmStateValue sFinished;
39
40      private final DFEsmDualPortMappedRAM sHistogram;
41
42      HistogramSMStateMachine(KernelLib owner, int histogramSize, int width) {
43          super(owner);
44
45          mHistogramSize = histogramSize;
46          mShiftAmount = 8 - MathUtils.bitsToAddress(mHistogramSize);
47          int addressWidth = MathUtils.bitsToAddress(mHistogramSize);
48
49          // Types
50          mAddressType = dfeUInt(addressWidth);
51          mDataType = dfeUInt(width);
52
53          // I/O
54          ilImage = io.input("inImage", mDataType);
55          ilImageSize = io.scalarInput("imageSize", mDataType);
56
57          // State
58          sInputValidReg = state.value(dfeBool(), false);
59          sAddressReg = state.value(mAddressType, 0);
60          sInputCounter = state.value(mDataType, 0);
61          sFinished = state.value(dfeBool(), false);
62          sAddValue = state.value(mDataType, 0);
63
64          sHistogram = mem.ramMappedDualPort(
65              "histogram",
66              mDataType,
67              mHistogramSize,
68              DualPortRAMMode.READ_ONLY,
69              DualPortRAMMode.WRITE_ONLY,
70              Latency.ONE_CYCLE);
71      }
72

```

Listing 14: Histogram state machine part 2: Next State Function(HistogramSMStateMachine.maxj).

```

74  @Override
75  protected void nextState() {
76      // Store whether the current input is valid
77      sInputValidReg.next <== io.isInputValid();
78
79      // Set off a read from the RAM for the bucket where the current input falls
80      sHistogram.addressA <== (ilimage>>mShiftAmount).cast(mAddressType);
81
82      // Store the address that we started the read for
83      sAddressReg.next <== (ilimage>>mShiftAmount).cast(mAddressType);
84
85      // By default, we won't write
86      sHistogram.writeEnableB <== false;
87
88      // If the input was valid in the previous cycle
89      IF (sInputValidReg == true)
90      {
91          /*
92           * Check if the current input is different to the previous input, or
93           * if we have read all the data.
94           *
95           * If so, then we enable the write to the RAM.
96           */
97          IF (sAddressReg != (ilimage>>mShiftAmount).cast(mAddressType) | sFinished)
98          {
99              sHistogram.writeEnableB <== true;
100              sAddValue.next <== 0;
101          }
102          // Otherwise, we hold the write to the RAM and increment the amount we will add
103          ELSE
104          {
105              sHistogram.writeEnableB <== false;
106              sAddValue.next <== sAddValue+1;
107          }
108      }
109
110      // Always write out the data and address, as this has no effect without the write enable
111      sHistogram.dataInB <== sHistogram.dataOutA+sAddValue+1;
112      sHistogram.addressB <== sAddressReg;
113
114      // Count the amount of data we have read
115      IF (io.isInputValid() == true)
116          sInputCounter.next <== sInputCounter+1;
117
118      // Register the fact that we have read all the data
119      IF (sInputCounter >= ilimageSize-1)
120          sFinished.next <== true;
121  }
122
123
124  @Override
125  protected void outputFunction() {
126  }
127  }

```

*Listing 15:* Main function from CPU code for histogram example showing setting and reading back mapped memory contents using the Basic Static SLiC Interface (HistogramSMCPUCode.c).

```

69  int main()
70  {
71      uint32_t *inImage;
72      int width = 0, height = 0;
73
74      int histogramSize = 32;
75
76      if (histogramSize < 2 || !((histogramSize & (histogramSize - 1)) == 0)) {
77          fprintf (stderr, "histogramSize must be a power of two and >1!");
78          return 1;
79      }
80
81      loadImage(
82          "../ EngineCode/src/histogramsm/test2.ppm",
83          &inImage,
84          &width,
85          &height,
86          1);
87
88      size_t dataSize = width * height * sizeof(uint32_t);
89
90      // Allocate a buffer for the edge data to go into.
91      uint32_t *outImage = malloc(dataSize);
92      uint64_t *ramIn = malloc(32 * sizeof(uint64_t));
93      uint64_t* outHistogram = malloc(histogramSize * sizeof(uint64_t));
94      for (int i = 0; i < histogramSize; i++)
95          ramIn[i] = 0;
96
97      uint64_t* expectedHistogram = malloc(histogramSize * sizeof(uint64_t));
98      HistogramSMCPU(width * height, inImage, NULL, NULL, expectedHistogram);
99
100     printf ("Running DFE.\n");
101     HistogramSM(width * height, inImage, outImage, ramIn, outHistogram);
102
103     writeImage("test_output.ppm", outImage, width, height, 1);
104
105     int status = check(histogramSize, expectedHistogram, outHistogram);
106
107     if (status)
108         printf ("Test failed.\n");
109     else
110         printf ("Test passed OK!\n");
111
112     return status;
113 }

```

## 9 Debugging

You can debug state machines in a similar way to Kernels using the `debug.simPrintf` statement in the Next State Function or Output Function.

Example 6 has the state machine from Example 2 with debug statements added to the Next State Function, shown in [Listing 16](#). The first `debug.simPrintf` statement outputs the current state of some state variables, including an extra counter to keep track of the current cycle count, every cycle:

```

58     sCycleCounter.next <== sCycleCounter + 1;
59     debug.simPrintf("Cycle %d: m_mode=%s, m_max=%d, m_counter=%d, m_holdCounter=%d, m_maxCounter=%d, io.isInputValid
    =%d\n",
60         sCycleCounter, sMode, iMax, sCounter, sHoldCounter, sMaxCounter, io.isInputValid());

```

A `debug.simPrintf` statement can be conditionally enabled by placing it inside an IF or SWITCH conditional statement. For example, a line is printed to the output if the current counter has hit the specified maximum count:

```

61     SWITCH(sMode) {
62         CASE(Modes.COUNTING_UP) {
63             IF(sCounter == iMax) {
64                 debug.simPrintf("Reached maximum\n");

```

By default, the output of `debug.simPrintf` statements is both displayed via standard output to the console and directed to a text file `debug/debug_printf.txt` created in the run directory. The name of the debug file can be changed, and the console output suppressed, via the SLiC configuration settings (see the Multiscale Dataflow Programming document for more detail on SLiC configuration).

### 9.1 Identifying the State Machine Instance

If more than one copy of a state machine is instantiated in a Kernel or the same state machine is used in multiple Kernels, there is a special string that you can insert into a simulation `printf` statement to identify the name of the instance producing the output: `"%StateMachine%"`. For example:

```
debug.printf("State Machine Instance: %StateMachine%, Cycle %d: sMode=%s, sMax=%d", sCycleCounter, sMode, sMax);
```

*Listing 16:* Program for a state machine demonstrating the use of debug.simPrintf (DebugSMStateMachine.maxj).

```

56  @Override
57  protected void nextState() {
58      sCycleCounter.next <= sCycleCounter + 1;
59      debug.simPrintf("Cycle %d: m_mode=%s, m_max=%d, m_counter=%d, m_holdCounter=%d, m_maxCounter=%d, io.isInputValid
        =%d\n",
60          sCycleCounter, sMode, iMax, sCounter, sHoldCounter, sMaxCounter, io.isInputValid());
61      SWITCH(sMode) {
62          CASE(Modes.COUNTING_UP) {
63              IF(sCounter == iMax) {
64                  debug.simPrintf("Reached maximum\n");
65                  sMode.next <= Modes.HOLD_COUNT;
66                  IF (io.isInputValid ())
67                      sMaxCounter.next <= sMaxCounter+1;
68              } ELSE {
69                  IF(sCounter > iMax) {
70                      debug.simPrintf("Reverted to new maximum\n");
71                      sCounter.next <= iMax;
72                      sMode.next <= Modes.HOLD_COUNT;
73                      IF (io.isInputValid ())
74                          sMaxCounter.next <= sMaxCounter+1;
75                  } ELSE {
76                      sCounter.next <= sCounter + 1;
77                  }
78              }
79          }
80          CASE(Modes.COUNTING_DOWN) {
81              IF(sCounter == 0) {
82                  sCounter.next <= sCounter + 1;
83                  sMode.next <= Modes.COUNTING_UP;
84              } ELSE
85                  sCounter.next <= sCounter - 1;
86          }
87          OTHERWISE {
88              IF (sHoldCounter == iHoldCount)
89              {
90                  sHoldCounter.next <= 0;
91                  sMode.next <= Modes.COUNTING_DOWN;
92              }
93              ELSE
94                  sHoldCounter.next <= sHoldCounter+1;
95          }
96      }
97  }

```

## 10 Modularizing Code

Within the Next State Function and Output Function, you can call any Java code, including other methods on the same class or instantiating other classes and invoking their methods. In this way, a complex state machine can be modularized effectively.

### 10.1 Intermediate Values

State values (`DFEsmStateValue`) are subclasses of `DFEsmValue`. All operators on an (`DFEsmStateValue`) return an `DFEsmValue` reference giving access to the current value. An `DFEsmValue` reference does not have state. An `DFEsmValue` can be referred to as an *intermediate* value that only holds its value for the current cycle.

In most cases, you will not need to explicitly create an `DFEsmValue` instance, so the distinction can be considered an implementation detail. In some cases an explicit `DFEsmValue` instance is required, for example to return a state machine value from a Java method.

An instance of `DFEsmValue` takes the same types as an `DFEsmStateValue` and has the same overloaded operators.

The following simple example shows a function implementing an adder being used with different arguments:

```
DFEsmStateValue a, b, c, d;  
...  
DFEsmValue DoAdd(DFEsmStateValue a, DFEsmStateValue b) {  
    return a+b;  
}  
...  
c.next <== DoAdd(a,b);  
d.next <== DoAdd(b,c);
```

### 10.2 State Machine Libraries

A state machine library allows you to implement state machine functionality independently from a state machine instance, for example to create functionality that can be used in a variety of both Kernel and Manager state machines.

Methods provided by the state machine library can take state variables, intermediate values and state machine inputs and outputs as arguments.



A method that assigns to an output cannot be called from the Next State Function of a state machine and, likewise, a method that assigns to a state variable cannot be called from the Output Function. Both cases will cause an exception to be raised by MaxCompiler.

A state machine library can have its own member state variables, which must be declared at the class level and initialized when an instance of the class is created, and also update state variables passed in as arguments to its methods.

A state machine library does not have its own Next State Function, so no member state variables will be updated in a cycle unless a method that writes to these member state variables is called from the Next State Function of a state machine. A state machine library also does not have an Output Function and you cannot declare state machine inputs and outputs.



### 10.3 State Machine Library Example

Example 7 takes the simple counter from Exercise 1 and implements the counting logic as a state machine library. This library is then used in both a Manager state machine and a Kernel state machine.

#### 10.3.1 The State Machine Library

The full source for the state machine library is shown in [Listing 17](#).

The declaration of a state machine library is similar to a state machine, except that the class must extend `StateMachineLib`:

```
20 class ModularSMStateMachineLib extends StateMachineLib {
```

The state variables from the Kernel state machine in Example 1 have been moved into this state machine library:

```
27 // State
28 private final DFESmStateValue sHoldCounter;
29 private final DFESmStateValue sCounter;
30 private final DFESmStateEnum<Modes> sMode;
```

These state variables are now initialized in the constructor of the state machine library:

```
32 ModularSMStateMachineLib(StateMachine parent, int width, int holdCount) {
33     super(parent);
34     m_holdCount = holdCount;
35     // State
36     DFESmValueType counterType = dfcUInt(width);
37     sMode = state.enumerated(Modes.class, Modes.COUNTING_UP);
38     sHoldCounter = state.value(counterType, 0);
39     sCounter = state.value(counterType, 0);
40 }
```

The logic for updating the counter has also been moved into a method called `updateCounter`, which takes an intermediate value, `iMax` as an argument:

```
42 void updateCounter(DFESmValue iMax)
43 {
44     sCounter.next <= sCounter;
45     SWITCH(sMode) {
46         CASE(Modes.COUNTING_UP) {
47             IF(sCounter == iMax) {
48                 sMode.next <= Modes.HOLD_COUNT;
49             } ELSE {
50                 /* We need to check that we are not larger than
51                  * m_max, as it might have changed
52                  */
53                 IF(sCounter > iMax) {
54                     sCounter.next <= iMax;
55                     sMode.next <= Modes.HOLD_COUNT;
```

`sCounter` is updated by the method every cycle in which it is called by a state machine. The method `getCounter` returns the current value of the counter:

```
82 DFESmValue getCounter() {
83     return sCounter;
84 }
```

Listing 17: State machine library example (ModularSMStateMachineLib.maxj).

```

11 package modularism;
12
13 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateEnum;
14 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateValue;
15 import com.maxeler.maxcompiler.v2.statemachine.DFEsmValue;
16 import com.maxeler.maxcompiler.v2.statemachine.StateMachine;
17 import com.maxeler.maxcompiler.v2.statemachine.StateMachineLib;
18 import com.maxeler.maxcompiler.v2.statemachine.types.DFEsmValueType;
19
20 class ModularSMStateMachineLib extends StateMachineLib {
21     enum Modes {
22         COUNTING_UP, HOLD_COUNT, COUNTING_DOWN
23     }
24
25     private final int m_holdCount;
26
27     // State
28     private final DFEsmStateValue sHoldCounter;
29     private final DFEsmStateValue sCounter;
30     private final DFEsmStateEnum<Modes> sMode;
31
32     ModularSMStateMachineLib(StateMachine parent, int width, int holdCount) {
33         super(parent);
34         m_holdCount = holdCount;
35         // State
36         DFEsmValueType counterType = dfeUInt(width);
37         sMode = state.enumerated(Modes.class, Modes.COUNTING_UP);
38         sHoldCounter = state.value(counterType, 0);
39         sCounter = state.value(counterType, 0);
40     }
41
42     void updateCounter(DFEsmValue iMax)
43     {
44         sCounter.next <= sCounter;
45         SWITCH(sMode) {
46             CASE(Modes.COUNTING_UP) {
47                 IF(sCounter == iMax) {
48                     sMode.next <= Modes.HOLD_COUNT;
49                 } ELSE {
50                     /* We need to check that we are not larger than
51                      * m_max, as it might have changed
52                      */
53                     IF(sCounter > iMax) {
54                         sCounter.next <= iMax;
55                         sMode.next <= Modes.HOLD_COUNT;
56                     } ELSE {
57                         sCounter.next <= sCounter + 1;
58                     }
59                 }
60             }
61             CASE(Modes.COUNTING_DOWN) {
62                 IF(sCounter == 0) {
63                     sCounter.next <= sCounter + 1;
64                     sMode.next <= Modes.COUNTING_UP;
65                 } ELSE {
66                     sCounter.next <= sCounter - 1;
67                 }
68             }
69             OTHERWISE {
70                 IF (sHoldCounter == m_holdCount)
71                 {
72                     sHoldCounter.next <= 0;
73                     sMode.next <= Modes.COUNTING_DOWN;
74                 }
75                 ELSE {
76                     sHoldCounter.next <= sHoldCounter+1;

```

Listing 18: Kernel state machine using a state machine library (ModularSMKernelStateMachine.maxj).

```

1  /**
2   * Document: MaxCompiler State Machine Tutorial (maxcompiler-sm-tutorial.pdf)
3   * Example: 7      Name: Modular state-machine
4   * MaxFile name: ModularSM
5   * Summary:
6   *   Kernel state machine using a state machine library which implements the
7   *   logic for a counter.
8   */
9  package modularsm;
10 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelLib;
11 import com.maxeler.maxcompiler.v2.statemachine.DFEsmInput;
12 import com.maxeler.maxcompiler.v2.statemachine.DFEsmOutput;
13 import com.maxeler.maxcompiler.v2.statemachine.kernel.KernelStateMachine;
14 import com.maxeler.maxcompiler.v2.statemachine.types.DFEsmValueType;
15
16 class ModularSMKernelStateMachine extends KernelStateMachine {
17
18     private final DFEsmOutput oCount;
19     private final DFEsmInput iMax;
20
21     private final ModularSMStateMachineLib mySMLib;
22
23     ModularSMKernelStateMachine(KernelLib owner, int width, int holdCount) {
24         super(owner);
25         mySMLib = new ModularSMStateMachineLib(this, width, holdCount);
26         DFEsmValueType counterType = dfeUInt(width);
27         // I/O
28         oCount = io.output("count", counterType);
29         iMax = io.input("max", counterType);
30     }
31
32     @Override
33     protected void nextState() {
34         mySMLib.updateCounter(iMax);
35     }
36
37     @Override
38     protected void outputFunction() {
39         oCount <= mySMLib.getCounter();
40     }
41 }

```

### 10.3.2 The Kernel State Machine

The Kernel state machine that uses this state machine library is shown in [Listing 18](#).

The counter logic has been removed from the state machine now that it is implemented in the state machine library. An instance of the state machine library is declared as a class member:

```

21     private final ModularSMStateMachineLib mySMLib;

```

This instance is initialized in the Kernel state machine constructor:

```

23     ModularSMKernelStateMachine(KernelLib owner, int width, int holdCount) {
24         super(owner);
25         mySMLib = new ModularSMStateMachineLib(this, width, holdCount);

```

The `updateCounter` method on the state machine library instance is now called from the `Next`

State Function with the input to the Kernel as the argument :

```

33  protected void nextState() {
34      mySMLib.updateCounter(iMax);
35  }

```

This method is called every cycle, which updates the counter.

The Output Function simply connects the output of `getCounter` to the output:

```

38  protected void outputFunction() {
39      oCount <= mySMLib.getCounter();
40  }

```

### 10.3.3 The Manager State Machine

The Manager state machine that uses this state machine library is shown in [Listing 19](#) and [Listing 20](#). This state machine has a pull input and a push output.

An instance of the state machine library is declared as a class member:

```

29  private final ModularSMStateMachineLib mySMLib;

```

This instance is initialized in the Manager state machine constructor:

```

31  ModularSMManagerStateMachine(DFEManager owner, int width, int holdCount) {
32      super(owner);
33
34      mySMLib = new ModularSMStateMachineLib(this, width, holdCount);

```

The Next State Function conditionally calls the `updateCounter` method on the state machine library whenever there is new data to process and sets the output to valid:

```

58      sOutputValid.next <= false;
59      IF (sReadDataReady == true) {
60          mySMLib.updateCounter(iMax);
61          sOutputValid.next <= true;
62          sOutputData.next <= mySMLib.getCounter();
63      }

```

*Listing 19: Manager state machine using a state machine library (ModularSMMManagerStateMachine.maxj).*

```

10 package modularsm;
11
12 import com.maxeler.maxcompiler.v2.managers.DFEManager;
13 import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateValue;
14 import com.maxeler.maxcompiler.v2.statemachine.manager.ManagerStateMachine;
15 import com.maxeler.maxcompiler.v2.statemachine.manager.DFEsmPullInput;
16 import com.maxeler.maxcompiler.v2.statemachine.manager.DFEsmPushOutput;
17
18 final class ModularSMMManagerStateMachine extends ManagerStateMachine {
19     // inputs/outputs
20     private final DFEsmPullInput iMax;
21     private final DFEsmPushOutput oOutput;
22
23     // state
24     private final DFEsmStateValue sReadDataReady;
25
26     private final DFEsmStateValue sOutputData;
27     private final DFEsmStateValue sOutputValid;
28
29     private final ModularSMStateMachineLib mySMLib;
30
31     ModularSMMManagerStateMachine(DFEManager owner, int width, int holdCount) {
32         super(owner);
33
34         mySMLib = new ModularSMStateMachineLib(this, width, holdCount);
35
36         sReadDataReady = state.value(dfeBool(), false);
37
38         sOutputValid = state.value(dfeBool(), false);
39         sOutputData = state.value(dfeUInt(32));
40
41         // Set up inputs/outputs
42         iMax = io.pullInput("max", dfeUInt(32));
43         oOutput = io.pushOutput("count", dfeUInt(32), 1);
44     }

```

*Listing 20: Manager state machine using a state machine library (ModularSMMManagerStateMachine.maxj).*

```

46  @Override
47  protected void nextState() {
48      /*
49       * We only get data from the input if :
50       * - the input was not empty in the previous cycle
51       * AND
52       * - the output was not stalled in the previous cycle
53       */
54      sReadDataReady.next <== ~iMax.empty & ~oOutput.stall;
55
56      /*
57       * If we get data from the input, then we can update the counter */
58      sOutputValid.next <== false;
59      IF (sReadDataReady == true) {
60          mySMLib.updateCounter(iMax);
61          sOutputValid.next <== true;
62          sOutputData.next <== mySMLib.getCounter();
63      }
64  }
65
66  @Override
67  protected void outputFunction() {
68      /*
69       * We only request data from the input if :
70       * - the input is not empty
71       * AND
72       * - the output is not stalled
73       */
74      iMax.read <== ~iMax.empty & ~oOutput.stall;
75
76      // Output the counter value, if it is valid
77      oOutput.valid <== sOutputValid;
78      oOutput <== sOutputData.cast(oOutput.getType());
79  }

```

## 11 Exercises

This section has a number of exercises for you to use some of the features that have been covered in this document.

### 11.0.4 Exercise 1: Morse code decoding using a ROM

In this exercise, you will take the Morse code tokenizer from Example 2 and extend it to decode the tokens into ASCII values using a ROM within the state machine.

The decoding from a sequence of dots and dashes can be achieved by traversing a binary tree of characters: each time a dot is encountered, the left branch is taken and each time a dash is encountered, the right branch is taken, as shown in [Figure 6](#).

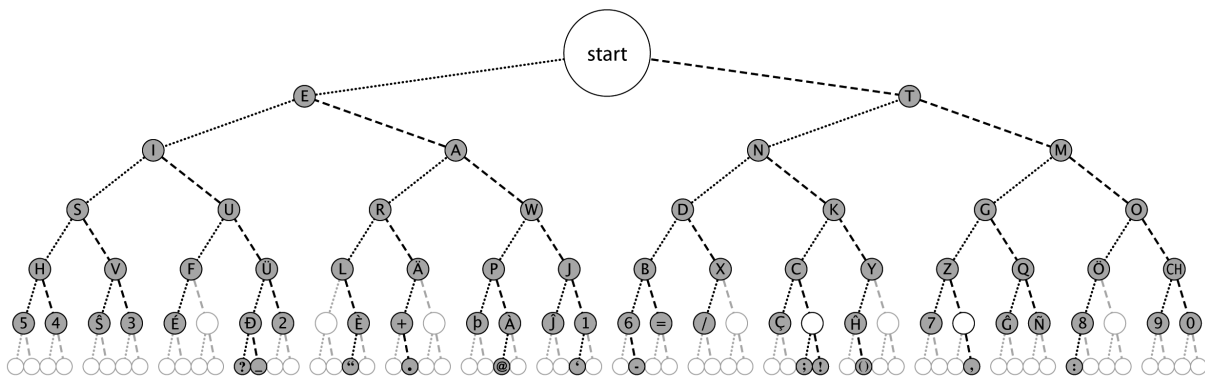


Figure 6: Binary tree for Morse code, courtesy of Wikipedia (<http://en.wikipedia.org/wiki/User:Aris00>) under the Creative Commons license CC-BY-SA (<http://creativecommons.org/licenses/by-sa/3.0/>).

This binary tree can be implemented in software using an array. Branching is achieved by doubling the current index into the array and adding 1 for the left branch or 2 for the right branch. The first element in the array is a space ( ' ' ). The following pseudo code shows the algorithm:

```
array[] = {
    ' ',
    'E', 'T',
    'I', 'A', 'N', 'M',
    'S', 'U', 'R', 'W', 'D', 'K', 'G', 'O',
    'H', 'V', 'F', '?', 'L', '?', 'P', 'J', 'B', 'X', 'C', 'Y', 'Z', 'Q', '?', '?',
    '5', '4', '?', '3', '?', '?', '?', '2', '?', '?', '+', '?', '?', '?', '1',
    '6', '=', '/', '?', '?', '?', '?', '?', '7', '?', '?', '?', '8', '?', '9', '0',
    '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?',
    '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?',
    '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?',
    '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?', '?'
};

index = 0;

for each token
```

```

switch (token)
  case (dot): index = 2*index + 1;
  case (dash): index = 2*index + 2;
  case (done_letter): output = array[index]; index = 1;
  case (done_word): output = array[0]; index = 1;
end switch
end for

```

For simplicity, this code only represents the Latin alphabet (without diacritics) and punctuation characters; other characters and invalid characters are represented as a question mark ('?').

Use this approach to add a stage that takes the token produced by the tokenizer, decodes this using a ROM and then writes the retrieved character to the output.

The source to the original Tokenizer is provided for you to modify and the C CPU code has been modified to test the output of your implementation.

### 11.0.5 Exercise 2: Run-Length decoding in a Manager state machine

In this exercise, you will implement run-length encoding on an input image in a Manager state machine.

Run-length encoding is a simple, loss-less form of compression where sequential values that are identical are compressed to a value and a count of how many times the value occurs in the sequence.

This input image is a stream of 24-bit unsigned integers representing the RGB pixel values. Your state machine needs to encode the output as a stream of 32-bit unsigned integers, where the lower 24-bits represent the RGB pixel value and the top 8-bits are a count of how many times that value appears in succession.



The input stream is padded to 32-bit values with 8 bits of padding that can be discarded, to make the software implementation clearer. The output stream is padded to 64-bit values to avoid packing to the CPU-interface word length (64 bits on a MAX2 and 128 bits on a MAX3).

The algorithm for a software implementation iterating over an array representing the image is shown below:

```

int image[image_size];
run_count = 1;
for (i = 1; i < image_size; i++)
  if (image[i] == image[i-1])
    run_count++;
  else
    output (image[i-1], run_count);
    run_count = 1;
  end if
end for
output (image[image_size-1], run_count);

```

Note that the first and last pixels need special treatment to get the correct count for the first pixel and to push the last value out.

A starting point of a state machine which simply passes the input to the output is provided. The CPU test code provided decodes the output from the state machine and compares it to the input image.