Introduction to State-based Applications in LabVIEW

Hans-Petter Halvorsen, 2016.10.31

http://home.hit.no/~hansha
Preface

Statecharts or State Diagrams are useful in simplifying the design of applications that use complex decision-making algorithms. By constructing a statechart, you can visualize the flow of a complex decision-making algorithm and achieve a high-level view of an application.

This document explains the basic concepts of creating state-based applications in LabVIEW.

You should have some basic knowledge about LabVIEW, e.g., the “An Introduction to LabVIEW” training. This document is available for download at http://home.hit.no/~hansa/.

In addition to LabVIEW Professional Development System, you need to install the “LabVIEW Statechart Module”.
Table of Contents

Preface .............................................................................................................................................. 2

Table of Contents ............................................................................................................................. iii

1 Introduction to LabVIEW .................................................................................................................. 1

1.1 Dataflow programming ................................................................................................................ 1

1.2 Graphical programming .............................................................................................................. 2

1.3 Benefits ........................................................................................................................................ 2

2 Introduction to Sequential Control ................................................................................................. 3

3 Introduction to State Diagrams ........................................................................................................ 4

4 Introduction to Sequential Function Charts (SFC) ........................................................................ 5

5 State-machine with LabVIEW .......................................................................................................... 7

5.1 While Loop .................................................................................................................................. 7

5.2 Case Structure .............................................................................................................................. 8

5.3 Shift Register .............................................................................................................................. 9

5.4 State Programming Architecture ............................................................................................... 10

5.5 State-machine ........................................................................................................................... 11

6 A Sequential Control example in LabVIEW .................................................................................. 14

7 Introduction to LabVIEW Statechart Module ................................................................................ 16

7.1 Procedure for creating a Statechart application in LabVIEW .................................................. 17

7.2 Statechart Editor ....................................................................................................................... 21

7.2.1 Statechart Development objects ......................................................................................... 23

7.3 Using a Caller VI to Execute a Statechart ................................................................................ 25

8 Tutorial: Create a Statechart ......................................................................................................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Advanced Topics in LabVIEW Statechart Module</td>
<td>42</td>
</tr>
<tr>
<td>9.1</td>
<td>Using Regions and Substates</td>
<td>42</td>
</tr>
<tr>
<td>9.2</td>
<td>Using Orthogonal Regions and State History</td>
<td>42</td>
</tr>
<tr>
<td>9.3</td>
<td>Splitting and Merging Transition Segments</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Whats Next?</td>
<td>44</td>
</tr>
<tr>
<td>10.1</td>
<td>My Blog</td>
<td>44</td>
</tr>
<tr>
<td>10.2</td>
<td>Training</td>
<td>44</td>
</tr>
<tr>
<td>10.3</td>
<td>Additional Resources</td>
<td>44</td>
</tr>
<tr>
<td>10.4</td>
<td>Examples</td>
<td>44</td>
</tr>
<tr>
<td>10.5</td>
<td>Documentation</td>
<td>45</td>
</tr>
<tr>
<td>10.6</td>
<td>LabVIEW Wiki</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td>10.7</td>
<td>LabVIEW on YouTube</td>
<td>Error! Bookmark not defined.</td>
</tr>
<tr>
<td></td>
<td>Quick Reference</td>
<td>46</td>
</tr>
</tbody>
</table>
1 Introduction to LabVIEW

LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench) is a platform and development environment for a visual programming language from National Instruments. The graphical language is named "G". Originally released for the Apple Macintosh in 1986, LabVIEW is commonly used for data acquisition, instrument control, and industrial automation on a variety of platforms including Microsoft Windows, various flavors of UNIX, Linux, and Mac OS X. Visit National Instruments at www.ni.com.

The code files have the extension “.vi”, which is an abbreviation for “Virtual Instrument”. LabVIEW offers lots of additional Add-Ons and Toolkits.

This paper is part of a series with LabVIEW tutorials, such as:

- Introduction to LabVIEW
- Data Acquisition in LabVIEW
- Control and Simulation in LabVIEW
- LabVIEW MathScript
- Linear Algebra in LabVIEW
- Datalogging and Supervisory Control in LabVIEW
- Wireless Data Acquisition in LabVIEW
- Intermediate Topics in LabVIEW
- Advanced Topics in LabVIEW

Each tutorial may be used independently of each other.

1.1 Dataflow programming

The programming language used in LabVIEW, also referred to as G, is a dataflow programming language. Execution is determined by the structure of a graphical block diagram (the LV-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution. Multi-processing and multi-threading hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.
1.2 Graphical programming

LabVIEW ties the creation of user interfaces (called front panels) into the development cycle. LabVIEW programs/subroutines are called virtual instruments (VIs). Each VI has three components: a block diagram, a front panel, and a connector panel. The last is used to represent the VI in the block diagrams of other, calling VIs. Controls and indicators on the front panel allow an operator to input data into or extract data from a running virtual instrument. However, the front panel can also serve as a programmatic interface. Thus a virtual instrument can either be run as a program, with the front panel serving as a user interface, or, when dropped as a node onto the block diagram, the front panel defines the inputs and outputs for the given node through the connector pane. This implies each VI can be easily tested before being embedded as a subroutine into a larger program.

The graphical approach also allows non-programmers to build programs simply by dragging and dropping virtual representations of lab equipment with which they are already familiar. The LabVIEW programming environment, with the included examples and the documentation, makes it simple to create small applications. This is a benefit on one side, but there is also a certain danger of underestimating the expertise needed for good quality "G" programming. For complex algorithms or large-scale code, it is important that the programmer possess an extensive knowledge of the special LabVIEW syntax and the topology of its memory management. The most advanced LabVIEW development systems offer the possibility of building stand-alone applications. Furthermore, it is possible to create distributed applications, which communicate by a client/server scheme, and are therefore easier to implement due to the inherently parallel nature of G-code.

1.3 Benefits

One benefit of LabVIEW over other development environments is the extensive support for accessing instrumentation hardware. Drivers and abstraction layers for many different types of instruments and buses are included or are available for inclusion. These present themselves as graphical nodes. The abstraction layers offer standard software interfaces to communicate with hardware devices. The provided driver interfaces save program development time. The sales pitch of National Instruments is, therefore, that even people with limited coding experience can write programs and deploy test solutions in a reduced time frame when compared to more conventional or competing systems. A new hardware driver topology (DAQmxBase), which consists mainly of G-coded components with only a few register calls through NI Measurement Hardware DDK (Driver Development Kit) functions, provides platform independent hardware access to numerous data acquisition and instrumentation devices. The DAQmxBase driver is available for LabVIEW on Windows, Mac OS X and Linux platforms.
2 Introduction to Sequential Control

Sequential control is crucial in many control applications, as in chemical batch processes. A sequential control procedure can be represented graphically by e.g. one of the following two methods:

- A State diagram, also denoted a State machine
- A Sequential function chart (SFC)

State diagrams and SFC are quite similar, as they have the same basic elements:

- **States** with **actions** to be executed when the state is active
- **Transitions** from one active state to another

Sequential control can be implemented using graphical programming of state diagrams or SFCs in proper programming tools. For example, state diagrams are supported by the LabVIEW Statechart Module. Different PLC (Programmable Logic Controller) tools support SFC.
State diagrams have states with actions to be executed when the state is active. They have transitions from one active state to another taking place if the transition condition is logically true.

Below we see a typical state diagram:

A state diagram represents a state machine. States of a state machine are either active or passive. Only one state is active at a time. The state machine always starts in a particular state defined as the initial state, and it ends (stops) after the final state. Special symbols or colors may be used for the initial state and the final state. See the Figure above.
Introduction to Sequential Function Charts

Sequential Function Charts describe control sequences in graphical form. State Diagrams and Sequential function charts (SFC) techniques are very similar. They have **states/steps** (in SFC the term steps is used) with **actions** to be executed when the state is active. They have **transitions** from one active state to another taking place if the transition condition is logically true.

Sequential function chart (SFC) is a graphical programming language often used for PLCs (PLC - Programmable Logic Controller).

Below we see a typical SFC chart:

![Sequential Function Chart](http://www.plcdev.com/sequential_function_charts_all)

The Sequential Function Charts break a sequential task down into Steps, Transitions and Actions. These are drawn graphically to describe a sequence of interactions as seen above.

**Steps:**
- Every step can execute one or more actions.
- A step can be either active or inactive. When it becomes active, it's actions are executed
Transitions:
- There are transitions between every step.
- When a Step is active and the following transition become true the successor step becomes active and the preceding step becomes inactive.
- Each transition has associated events which are called transition conditions.
- The transition conditions must combine to make a simple Boolean result, True or False.

Step Actions:
- Every action is associated with a step. The step can have none or several actions associated.
- If there is no associated action to the step, it will be considered as a WAIT function.
- Each step/action association shall have an associated action qualifier.

The Program Flow is as follows:
- A step becomes active when its preceding transition activates (that is the transition is true, while the step before that transition is active.)
- A step becomes inactive when it's subsequent transition becomes true
- In Parallel branches, all the initial steps become active when the preceding transition activates
- A parallel branch end when all the steps before the branch end are be active, and the subsequent transition becomes true

Many systems have sequential operation requirements and Sequential Function Charts (SFC’s) have become a popular method of accurately specifying sequential control requirements. SFC’s have many advantages for software development both in the design stage as well as the implementation and testing, maintaining and fault finding stages, such as:

- Design Stage
  - Detailed clear graphical specification
  - Non software people can specify / verify
- Implementation
  - Straight forward conversion from specification to code
  - Structured testing / debugging
- Maintenance of Software
  - Readily understood by engineer modifying software
- Machine Maintenance
  - Allows quick accurate fault diagnosis
5 State-machine with LabVIEW

The State Machine approach in LabVIEW uses a Case structure inside a While loop to handle the different states in the program, and the transitions between them. The Shift Register is used to save data from and between the different states.

Below we shortly repeat the While Loop, the Case structure and the Shift Register before we dig into the principles of the state-machine.

For more details about basic LabVIEW programming techniques, refer to the tutorial “Introduction to LabVIEW”. The tutorial is available for download from:
http://home.hit.no/~hansha.

5.1 While Loop

A While loop repeats the sub diagram inside it until the conditional terminal, an input terminal, receives a particular Boolean value. The Boolean value depends on the continuation behavior of the While Loop. Right-click the conditional terminal and select Stop if True or Continue if True from the shortcut menu. You also can wire an error cluster to the conditional terminal, right-click the terminal, and select Stop on Error or Continue while Error from the shortcut menu. Note! The While Loop always executes at least once.

Below we see an empty While loop:
After you create a While Loop, you can use **shift registers** to pass values from one iteration to the next. If you wire an array to a While Loop, you can read and process every element in that array by enabling **auto-indexing**. See example below.

![Shift Register Example](image)

In order to convert a While Loop into a For Loop, right-click the While Loop and select “Replace with For Loop” from the shortcut menu. To convert a While Loop into a Timed Loop, right-click the While Loop and select “Replace with Timed Loop” from the shortcut menu.

### 5.2 Case Structure

The Case Structure has one or more sub diagrams, or cases, exactly one of which executes when the structure executes. The value wired to the selector terminal determines which case to execute and can be Boolean, string, integer, or enumerated type. You may right-click the structure border to add or delete cases. Use the Labeling tool to enter value(s) in the case selector label and configure the value(s) handled by each case.

Below we see an empty Case structure:

![Case Structure Example](image)
Below we see an example of a Case structure with 2 cases, a “True” case and a “False” case. Depending of the Boolean input value, the Numeric1 and Numeric2 is either Added or Subtracted.

You may also connect an error cluster directly to a case structure, like this:

The case structure automatically switches to a “No Error” case and an “Error” case.

5.3 Shift Register
Use shift registers on For Loops and While Loops to transfer values from one loop iteration to the next. Shift registers are similar to static variables in text-based programming languages. A shift register appears as a pair of terminals, directly opposite each other on the vertical sides of the loop border. The right terminal contains an up arrow and stores data on the completion of an iteration. LabVIEW transfers the data connected to the right side of the register to the next iteration. Create a shift register by right-clicking the left or right border of a loop and selecting Add Shift Register from the shortcut menu.

A shift register transfers any data type and automatically changes to the data type of the first object wired to the shift register. The data you wire to the terminals of each shift register must be the same type.

To initialize a shift register, wire any value from outside the loop to the left terminal. If you do not initialize the shift register, the loop uses the value written to the shift register when the loop last executed or the default value for the data type if the loop has never executed.

Use a loop with an uninitialized shift register to run a VI repeatedly so that each time the VI runs, the initial output of the shift register is the last value from the previous execution. Use an uninitialized shift register to preserve state information between subsequent executions of a VI. After the loop executes, the last value stored in the shift register remains at the right terminal. If you wire the right terminal outside the loop, the wire transfers the last value stored in the shift register. You can add more than one shift register to a loop. If you have multiple operations within a loop, use multiple shift registers to store the data values from those different processes in the structure.

Shift Register Example:

![Shift Register Example Diagram](image)

Create and Run this example to see what happens.

### 5.4 State Programming Architecture
Creating VIs using the State Machine approach is very useful when creating (large) applications.

In general, a state machine is a model of behavior composed of a finite number of states, transitions between those states, and actions. It is similar to a "flow graph" where we can inspect the way in which the logic runs when certain conditions are met.

Sometimes, you may want to change the order of the sequence, repeat one item in the sequence more often than the other items, stop a sequence immediately, or have items in the sequence that may execute only when certain conditions are met. Although your program may not have any such requirements, there is always the possibility that the program must be modified in the future. Therefore, a state programming architecture is a good choice, even if a sequential programming structure is sufficient. The following list describes more complex programming requirements that justify the use of a state programming architecture for an application.

- You need to change the order of the sequence
- You must repeat an item in the sequence more often than other items
- You want some items in the sequence to execute only when certain conditions are met

### 5.5 State-machine

The State Machine approach in LabVIEW uses a Case structure inside a While loop to handle the different states in the program, and the transitions between them. The Shift Register is used to save data from and between the different states.

Below we see examples of a state machine principle implemented in LabVIEW.
Simple State Machine principle

State Machine with multiple transitions depending on the State:

More advanced State Machine using Shift Registers:
Templates for State-machines in LabVIEW are available for download at:
http://home.hit.no/~hansha
6A Sequential Control example in LabVIEW

This example illustrates how we may use the state-machine principle in LabVIEW to create a sequential control application.

This example uses a state machine for sequential control of the position of a bicycle.

The state-machine is implemented with:

- An overall Case-structure which defines the states and contains the actions of each state
- A Shift-register which remembers which state is the active state
- An internal Case-structure inside each state (Case in the overall Case-structure) together with proper Boolean comparison functions implementing the transition conditions.

The state-machine controls a simulated process (bicycle position) given by the following dynamic model:

$$\frac{dx}{dt} = v$$

Where $x$ is position and $v$ is speed. The speed is manipulated by the state-machine.

This model is implemented in a Simulation Loop which runs in parallel with the While loop. Data (variables) are exchanged between these loops using local variables.

The arrow on a Vertical Pointer Slide is replaced with a picture of a bicycle. The procedure to substitute any graphical part of an indicator or control on the Front panel is as follows: Right-click on the item (graphical part), e.g. the arrow of a Vertical Pointer Slide. Select Advanced / Customize in the menu that is opened. Click the Tool button in the toolbar of the new window that is opened. Right-click on the arrow on the slider, and select the picture you want. Etc.

How to use it?

Use this VI as a template for your own state-machine controller for any simulated dynamic system. To control a real (physical) process instead, just substitute the simulated process with read and/or write functions for analog and/or digital I/O (input/output).
A Sequential Control example in LabVIEW

The Front Panel looks like this:

![Front Panel Image]

The Block Diagram looks like this:

![Block Diagram Image]

In order to get the feeling with state machines, while loops, case structures and shift registers, try to reproduce the example.

The example is created by Finn Haugen, University College of Southeast Norway.
Introduction to LabVIEW Statechart Module

The NI LabVIEW Statechart Module provides a programming model that you can use to design applications at a higher level than was previously possible. With the LabVIEW Statechart Module, you can develop statechart diagrams, define behaviors with LabVIEW dataflow programming.

Statecharts expand the classic state diagram with the additional features of hierarchy, concurrency, and comprehensive actions.

LabVIEW statecharts are especially useful for programming event-response applications such as intricate user interfaces and advanced state machines used to implement dynamic system controllers, machine control logic, and digital communication protocols.

Using the LabVIEW Statechart Module involves the following two steps:

- Using the Statechart Editor window to build a statechart.
- Creating a VI that executes the statechart. This VI also sends information to, and receives information from, the statechart.

Building a Statechart

Building a statechart involves defining the following items:

- **States**
- **Actions** and static reactions these states execute
- **Transitions** between states
- **Actions** these transitions execute
- **Triggers**

When you build a statechart, you define unique states in which the statechart can be. You also define transitions between these states, when the statechart can take these transitions, and what actions these transitions can take. You then define triggers that cause the statechart to begin evaluating a certain transition. For example, consider a statechart that you use to control a chemical process. You can specify that the statechart moves from State 1 to State 2 after receiving the Valve Open trigger. States and transitions also have
associated actions, which you use to modify output and state data associated with the
statechart.

**Creating a VI that Executes the Statechart**

After you define the valid triggers and configure the statechart to react to those triggers, you
must send those triggers to the statechart. You send triggers to the statechart by creating a
VI, known as the **Caller VI**, to send triggers at a specific time. For example, you can configure
the caller VI to send the Stop trigger after you click the Stop front panel button.

The caller VI also sends input data to the statechart so the statechart can act on that data.
The caller VI receives data that the statechart modifies as a result of a state or transition
action. You then can use this data in other parts of the caller VI or send this data to other
VIs.

The following figure shows how a caller VI interacts with a statechart.

![Caller VI](image)

The figure above shows how the caller VI sends data to the statechart, which returns output
data to the caller VI. If the statechart is asynchronous, the caller VI also might send a trigger
to the statechart. Sending a trigger to the statechart causes the statechart to begin
executing.

### 7.1 Procedure for creating a Statechart application in LabVIEW

The procedure for creating a statechart application in LabVIEW with the LabVIEW Statechart
Module is as follows:

**Step1:** In the New dialog select “Statechart”.
**Step 2:** Name the Statechart

**Step 3:** The Project Explorer for the Statechart application appears:
As you see, the LabVIEW Statechart Module automatically generates a set of Vis and Controls.

**LabVIEW Statechart Palette:**

LabVIEW Statechart Module offers the following Functions and Objects in the Statechart palette:

**Statechart Communication Functions:**

Use the Statechart Communication functions to execute a statechart, send triggers to a statechart, and determine whether a particular state is active. You can place these functions on a block diagram.

Statechart Communication palette:
Run Statechart: Executes an instance of the linked statechart. You typically place this function inside a loop in the caller VI.

Example:

Send External Trigger: Sends a trigger to the external queue of an instance of the linked statechart. You can use this function in the caller VI or in a VI that runs parallel to the caller VI. See example above.

IsIn: Determines whether the specified state is active. An active state is one that a statechart is currently in. You can use this function in a guard, an action, or a subVI that a guard or action calls.

Send Internal Trigger: Sends a trigger to the internal queue of the statechart. You can use this function within only an action or in a subVI that an action calls.
Statechart Development Objects:

Use the Statechart Development objects, consisting of states, regions, “pseudostates”, and connectors, to construct statecharts. **Note! You can place these objects only on a statechart diagram.**

Statechart Development palette:

![Image of Statechart Development palette](image)

The different objects are explained below.

7.2 Statechart Editor

You use the **Statechart Editor** window to edit a statechart. You can display the Statechart Editor window in one of the following ways:

- Double-click the “Diagram.vi” item in the Project Explorer window of the statechart.
- Right-click the “Diagram.vi” item and select Open from the shortcut menu.
- Right-click a Run Statechart function that links to the statechart and select View » Statechart Diagram from the shortcut menu.

Statechart Editor Window:
The Statechart Editor window, which displays a statechart diagram, is similar to the LabVIEW window that displays a block diagram. In both of these windows, you can place objects from the Functions palette on the diagram, use the LabVIEW tools to manipulate objects, and connect objects together. However, the Statechart Editor window is different from the LabVIEW window in the following ways:

- You can place only Statechart Development objects on a statechart diagram. You cannot place LabVIEW VIs, functions, or structures directly on a statechart diagram.

- The LabVIEW block diagram window displays the Run button on the toolbar. The Statechart Editor window displays the “Generate Code” button on the toolbar. Clicking this button generates code for the statechart.
- The LabVIEW block diagram has a front panel window. The Statechart Editor window does not have a front panel window.
- On the LabVIEW block diagram, you use the wiring tool to create wires between VIs and functions. In the Statechart Editor window, you use the wiring tool to create transitions between states and other objects.
- The LabVIEW block diagram provides options for debugging VIs. The Statechart Editor window does not provide any debugging options. You debug statecharts by using a separate window that you can access while the statechart is running.
- The Statechart Editor window disables many LabVIEW pull-down menu items that do not apply to statecharts.

7.2.1 Statechart Development objects

Use the Statechart Development objects to construct statecharts. You can place these objects on a statechart diagram.

The Statechart Development objects consist of states, regions, and connectors.

The Statechart Development palette:

The Statechart Development objects are as follows:

**State** Defines a state, which is a unique condition in which the statechart can be. You must place all states within a region. Note! The top-level statechart diagram represents a region. All states must have at least one incoming transition.
**Initial** Specifies the first state that the enclosing region enters when the statechart enters that region. To specify this state, create a transition from the Initial pseudostate to the state that you want to enter first. You can place this pseudostate within only a region or at the top level of a statechart diagram. Each region must have one and only one Initial pseudostate.

**Terminal** Specifies a state that can terminate the execution of the enclosing region. To specify this state, create a transition from a state to a Terminal pseudostate. You can place this pseudostate within only a region. You can place more than one Terminal pseudostate in a region.

**Junction** Connects common elements of multiple transitions together. For example, you can use a Junction connector to specify that multiple transitions use the same action, even though each transition might have a different trigger. You can place this connector only in a region.

**Region** Defines the area in which you can place states and pseudostates. Each region must contain an Initial pseudostate and at least one substate. Note! The top-level statechart diagram represents a region. You can place regions within only a state.

**Fork** Splits one incoming transition segment into multiple outgoing segments. You can place this connector within only a state.

**Join** Merges multiple incoming transition segments into one outgoing segment. You can place this connector within only a state.
Port Indicates where a transition leaves a state or connector. Ports also indicate where a transition passes through a region. When you are editing a statechart, ports appear automatically as you create transitions. You can place ports manually only when you are viewing a state as a subdiagram.

Shallow History Specifies that, when the statechart leaves and then returns to a region, the statechart enters the highest-level substates that were active when the statechart left the region. To use the Shallow History pseudostate, place this pseudostate in a region and then create a transition from a state to this pseudostate.

Deep History Specifies that, when the statechart leaves and then returns to a region, the statechart enters the lowest-level substates that were active when the statechart left the region. To use the Deep History pseudostate, place this pseudostate in a region and then create a transition from a state to this pseudostate.

7.3 Using a Caller VI to Execute a Statechart

The caller VI is the VI you create that contains a “Run Statechart” function. This function creates and executes an instance of a statechart. Executing a statechart means sending input data to a statechart. You also can use the caller VI to send triggers to a statechart.

The caller VI has a different structure depending on whether the statechart is asynchronous or synchronous. The following sections provide information about these differences.

Executing asynchronous statecharts involves the following two processes:

- Executing the statechart by using the “Run Statechart” function.
- Sending triggers to the external queue by using the “Send External Trigger” function.

In this situation, the VI that contains the “Run Statechart” function is the caller VI. You can send triggers to the external queue from another VI or a parallel loop in the caller VI. The following figure shows a caller VI that both executes an asynchronous statechart and sends triggers to the external queue.
In the figure above, the Statechart Loop, which is a While Loop, executes the statechart. The Inputs control contains any input data of the types that you defined. The Outputs indicator returns output data from the statechart. The Instance Name constant specifies Statechart 1 as the name of the statechart instance.

The Trigger Loop, which can execute in parallel with the Statechart Loop, contains an Event structure. This Event structure handles events that occur within the VI. In this case, an event occurs when you click the Button front panel control. When you click this control, the Event structure detects this event and the Send External Trigger function sends the Trigger1 trigger to the Statechart 1 statechart instance.

Note! This example uses the caller VI to send triggers to an external queue. However, as long as the caller VI is running and the statechart is in memory, you can use any VI to send triggers to an external queue. This behavior means that you can place the Trigger Loop and Statechart Loop in separate VIs.

**Statechart Editor:**

Open the Statechart Diagram (in the Statechart Editor) as follows:

Right-click on the “Run Statechart” VI and select View → Statechart Diagram.
Project Explorer:

Open the Project Explorer as follows:

Right-click on the “Run Statechart” VI and select View → Statechart.
8 Tutorial: Create a Statechart

In this Tutorial you will learn how to create a statechart from scratch. In this example we will create the following application:

When you push the different buttons, the current state will be displayed in the “Current Activity” indicator

This Tutorial goes through the following steps:

- **Step 1**: Creating and Configuring the Statechart
- **Step 2**: Defining States
- **Step 3**: Creating a Transition
- **Step 4**: Creating a Trigger and Configuring a Transition
- **Step 5**: Creating and Configuring the Terminal Transitions
- **Step 6**: Creating and Configuring the Initial pseudostate and Transition
- **Step 7**: Defining the Type of Output Data
- **Step 8**: Defining the Type of State Data
- **Step 9**: Configuring a State Entry Action
- **Step 10**: Configuring the Action of a Transition
- **Step 11**: Duplicating the Transition Node
- **Step 12**: Generating Code for the Statechart
Let's start the Tutorial:

**Step 1: Creating and Configuring the Statechart**

Complete the following steps to create and configure the statechart.

1. Launch LabVIEW and display the Getting Started window.
2. Select File»New to launch the New dialog box.
3. Select Other Files»Statechart from the Create New tree.

4. Click the OK button. LabVIEW prompts you to save the statechart.
5. Enter the name “Tutorial” as the name of your statechart (Note! It has the ending .lvsc) and save the statechart in a convenient location on disk.
6. LabVIEW displays a Project Explorer window that contains the necessary support files. This project item represents the “Tutorial.lvsc” file in which LabVIEW stores statechart information.

7. Right-click the “Tutorial.lvsc” item and select Properties. LabVIEW launches the Properties dialog box.
8. Select the “Statechart Code Generation” item from the Category list. LabVIEW displays options relating to statechart code generation.

9. Select Asynchronous from the Usage pull-down menu. This action specifies that the statechart is asynchronous.

10. Click the OK button to save changes and return to the Project Explorer window.

**Step 2: Defining States**

The next step is creating the states that comprise the statechart. A state is a unique condition in which the statechart can be.

This statechart consists of three states: **Menu**, **Play**, and **Pause**. Complete the following steps to create these states.

1. In the Project Explorer window, double-click the Diagram.vi item to launch the **Statechart Editor** window. This window displays the statechart diagram. You build a statechart by placing statechart objects on the statechart diagram.

2. If you do not see the Functions palette, select View»Functions Palette to display this palette.
3. On this palette, select Statechart»Statechart Development to display the Statechart Development palette.

4. Click the State object.
5. Move the cursor over the statechart diagram. Click to place the top left corner of the state, drag the cursor diagonally to establish the size of the state, and click again to place the state on the statechart diagram.
6. Notice the state is labeled State. Double-click this label to enable text editing and enter “Menu” as the new label.
7. Move the cursor over the black border of the Menu state until the cursor changes to the Positioning tool.
8. Press the <Ctrl> key and click and drag the cursor to the right of the Menu state. This action creates copy of the state.
9. Label this new state “Play”.
10. Create another new state labeled “Pause”. Place this state to the right of the Play state.
11. Return to the Statechart Development palette and place a “Terminal” pseudostate below the Play state.

The diagram should look like this:

![Statechart Diagram]

Step 3: Creating a Transition

The next step is defining the ways in which the statechart can move between states. To define this behavior, you use transitions to connect one state to another visually. Complete the following steps to create transitions between two states.
1. Move the cursor over the right border of the Menu state and change the cursor to the Wiring tool.
2. Click the state border. As you move the cursor across the statechart diagram, LabVIEW draws a dotted line between the state and the Wiring tool.
3. Move the cursor over the left border of the Play state. A gray box appears under the cursor.
4. Click the border to finish the transition. LabVIEW creates a transition between the two states.

The statechart diagram should look like this:

![Statechart Diagram]

**Step 4: Creating a Trigger and Configuring a Transition**

The next step is specifying that the statechart takes the transition only after receiving a specific event, also called a trigger. By default, transitions are configured to react to the NULL trigger. However, the caller VI that executes this statechart does not send the NULL trigger. Therefore, you must configure this transition to react to a different trigger.

Each statechart has a list of triggers from which you can choose when configuring a transition. Configuring a transition involves creating a trigger and then associating that trigger with the transition. Complete the following steps to configure the transition.

1. Double-click the transition node between the Menu and Play states. This action launches the Configure Transition dialog box. You use this dialog box to configure transitions. The first tab in this dialog box is the Triggers/Groups tab. Notice that the NULL trigger is selected by default in the Triggers and Groups list. Every statechart has this trigger by default.
2. Click the “Edit Triggers and Groups” button to launch the Edit Triggers and Groups dialog box. You use this dialog box to create and modify triggers.

3. Click the Create Trigger button to create a new trigger, which appears in the Triggers and Groups list.

4. Enter “Play” as the new trigger name.

5. Click the OK button to save changes and return to the Configure Transition dialog box. Notice the new trigger appears in the Triggers list.

6. In the Triggers/Groups list, place a checkmark in the Play checkbox to specify that this trigger causes the statechart to take the transition.

7. Remove the checkmark from the NULL checkbox.

8. Click the OK button to save changes and return to the statechart diagram.

9. Save the statechart by selecting File»Save in the Statechart Editor window. You also can press the <Ctrl-S> keys.

Notice the blue rectangle on the transition node. This rectangle indicates that you have specified a non-NULL trigger for the transition. If you move the cursor over the transition node, the Context Help window displays the properties of the transition, including the trigger. If you do not see the Context Help window, press the <Ctrl-H> keys to display this window.

Create and configure the following transitions and triggers. Remember also to deselect the NULL trigger for each transition.
• From the “Play” state to the “Menu” state. Configure this transition to react to the “Menu” trigger.
• From the “Pause” state to the “Play” state. Configure this transition to react to the “Play” trigger.
• From the “Play” state to the “Pause” state. Configure this transition to react to the “Pause” trigger.

The statechart should look like this:

![Statechart Diagram]

**Step 5: Creating and Configuring the Terminal Transitions**

Create a transition from the “Menu” state to the “Terminal” pseudostate. Configure this transition to react only to a new trigger, Stop. This transition specifies that if the statechart is in the Menu state and receives the Stop trigger, the statechart terminates.

**Tip!** Right-click the transition node and select either Rotate or Reverse to ensure the node faces the proper direction.

Then, save the statechart by pressing the <Ctrl-S> keys.

The statechart diagram should now look like this:

![Statechart Diagram with Terminal Transition]
Step 6: Creating and Configuring the Initial pseudostate and Transition

Place an Initial pseudostate, located on the Statechart Development palette, to the left of the “Menu” state.

Create a transition between this pseudostate and the “Menu” state. Notice the Generate Code button no longer is broken.

The statechart should look like this:

Step 7: Defining the Type of Output Data

If you want the statechart to modify data, you first must define the type of data. This process involves creating the controls and/or indicators that represent the statechart data.

Complete the following steps to define the statechart output data. Later in this part of the tutorial, you configure the statechart to modify this data.

1. Double-click the Outputs.ctl item in the project Explorer. LabVIEW displays the Control Editor window for this type definition.
2. Notice the numeric control Numeric inside the Outputs cluster. Right-click this control and select Change to Indicator to change this control to a numeric indicator.
3. Right-click the Numeric indicator and select Replace»String Indicator. This action replaces the numeric indicator with a string indicator.
4. Double-click the String label and enter Current Activity as the new label.
5. Press the <Ctrl-S> keys to save this type definition.
6. Close the Control Editor window to return to the Project Explorer window.
**Step 8: Defining the Type of State Data**

State data is different from output data. State data is accessible from only the statechart, whereas output data also is accessible from a VI that calls the statechart. Use state data to define data types that other VIs do not need to access.

Complete the following steps to define the statechart state data.

1. In the Project Explorer window, double-click the `StateData.ctl` item.
2. Right-click the Data control and select Change to Indicator. This action replaces the numeric control with a numeric indicator.
3. Double-click the Data label and enter Counter as the new label.
4. Press the `<Ctrl-S>` keys to save this type definition.
5. Close the Control Editor window to return to the Project Explorer window.

**Step 9: Configuring a State Entry Action**

The next step is configuring each state to modify the Current Activity indicator that you defined earlier. You accomplish this task by defining an entry action for a state. An entry action is LabVIEW block diagram code that the statechart executes immediately after entering a state.
Complete the following steps to configure the Menu state to update this indicator.

1. Double-click the Diagram.vi item to display the **Statechart Editor** window.
2. Double-click the border of the Menu state. This action launches the **Configure State** dialog box.
   **Note!** When you configure a state, a wrench icon \(\equiv\) appears on the state (\(\) in the Statechart Editor window. The state also changes color.

3. Notice the Outputs cluster element on the right side of this block diagram. Click this cluster element and select Outputs » Current Activity. This element now displays Outputs.Current Activity, which indicates you can write to the value of the type definition you defined.
4. Move the cursor over the input of this cluster element until the cursor changes to the Wiring tool.
5. Right-click the input and select Create » Constant. LabVIEW places a string constant on the block diagram.
6. Enter Menu as the value of the constant.
Tutorial: Create a Statechart

7. This entry action code means that, immediately after entering the Menu state, the statechart changes the Current Activity indicator to read Menu. Later in this tutorial, you will display the value of this indicator in the VI that calls the statechart.

8. Click the OK button to save changes and return to the statechart diagram.

9. The next step is defining the “Play” and “Pause” states to update the Current Activity indicator appropriately. Configure the Entry Action for each state to update this indicator to “Playing” or “Paused”, respectively.

10. Save the statechart.

**Step 10: Configuring the Action of a Transition**

The transition from the “Menu” state to the “Terminal” pseudostate also causes a change in the activity of the statechart. Complete the following steps to configure this transition.

Double-click the transition node in between the “Menu” state and “Terminal” pseudostate. This action launches the **Configure Transition** dialog box.

Click the Action tab.

Write the block diagram code that changes the value of the “Current Activity” indicator to “Stopped”.

Click the OK button to save changes and return to the statechart diagram.

**Note!** The transition node you just configured now displays two blue rectangles. The second blue rectangle indicates the transition has a specified action.

**Step 11: Duplicating the Transition Node**

The next step is creating transitions from the Play and Pause states to the Terminal pseudostate. Instead of configuring additional transition nodes, you can duplicate the transition node you configured already. Complete the following steps to duplicate this transition node.

1. Press the <Ctrl> key and click and drag the transition node you just configured.
2. Move this node under the Play state.
3. Click the bottom border of the Play state to initiate a transition segment.
4. Move the cursor over the incoming tunnel of the transition node until LabVIEW displays a tip strip that reads Segment In.
5. Click again to complete the transition segment.
6. Create a transition segment from the other side of this node to the Terminal pseudostate.
7. Right-click the transition node and select either Rotate or Reverse to ensure the node faces the proper direction.
Repeat steps 1–7 to create an additional transition node from the Pause state to the Terminal pseudostate. All three states now have a transition to the Terminal pseudostate.

Save the statechart.

**Step 12: Generating Code for the Statechart**

The next step is generating code for the statechart. This code is contained in a single Run Statechart function that you place on the block diagram of the caller VI. Complete the following steps to generate code for this statechart.

1. Display the **Statechart Editor** window.
2. Click the “**Generate Code**” button , which is located on the toolbar of this window.
3. Close the Statechart Editor window.

**Note!** You also can generate code by right-clicking the .lvsc file in the Project Explorer window and selecting Generate Code from the shortcut menu.

**Step 13: Creating the Caller VI**

The next step is to create the caller VI to execute and send triggers to the statechart. This VI contains two functions, **Run Statechart** and **Send External Trigger**, that you must link to the statechart.

**Run Statechart** Executes an instance of the linked statechart. You typically place this function inside a loop in the caller VI.

**Send External Trigger** Sends a trigger to the external queue of an instance of the linked statechart. You can use this function in the caller VI or in a VI that runs parallel to the caller VI.

Complete the following steps.

1. Select File»New VI from the Project Explorer window
2. Create 2 While Loops called “Statechart Loop” and “Trigger Loop”
3. Add the “Run Statechart” VI in the “Statechart Loop”
4. Add “Send External Trigger” function in the “Trigger Loop”
5. Link “Run Statechart” and “Send External Trigger” to the statechart. Right-click and select “Link to statechart...”
6. Finish the Caller VI so it looks like this:

7. The Front Panel should look like this:
8. Press the <Ctrl-R> keys to run the caller VI. Click the front panel buttons and notice how the Current Activity indicators changes according to the state.

**Step 14: Debugging**

Right-click the Run Statechart function and select “Debug Statechart...”. This action displays the statechart in the Statechart Debugging window.

Click the “Highlight Execution” button, located on the toolbar of the Statechart Debugging window, to enable statechart execution highlighting.

Run the Caller VI and see how the debugging works.
9 Advanced Topics in LabVIEW Statechart Module

In this chapter we will mention some of the more advance features in the LabVIEW Statechart Module.

9.1 Using Regions and Substates

We recall the following objects from the Statechart Development palette:

- **Region** Defines the area in which you can place states and pseudostates. Each region must contain an Initial pseudostate and at least one substate. The top-level statechart diagram represents a region. **You can place regions within only a state.**

- **Port** Indicates where a transition leaves a state or connector. Ports also indicate where a transition passes through a region. When you are editing a statechart, ports appear automatically as you create transitions. You can place ports manually only when you are viewing a state as a subdiagram.

9.2 Using Orthogonal Regions and State History

We recall the following objects from the Statechart Development palette:

- **Shallow History** Specifies that, when the statechart leaves and then returns to a region, the statechart enters the highest-level substates that were active when the statechart left the region. To use the Shallow History pseudostate, place this pseudostate in a region and then create a transition from a state to this pseudostate.

- **Deep History** Specifies that, when the statechart leaves and then returns to a region, the statechart enters the lowest-level substates that were active when the statechart left
the region. To use the Deep History pseudostate, place this pseudostate in a region and then create a transition from a state to this pseudostate.

9.3 Splitting and Merging Transition Segments

We recall the following objects from the Statechart Development palette:

**Fork** Splits one incoming transition segment into multiple outgoing segments. You can place this connector within only a state.

![Fork Diagram]

**Join** Merges multiple incoming transition segments into one outgoing segment. You can place this connector within only a state.

![Join Diagram]
10  Whats Next?

10.1  My Blog

For more information about LabVIEW, visit my Blog:  http://home.hit.no/~hansha/

10.2  Training

This Tutorial is a part of a series with other Tutorials I have made, such as:

- Introduction to LabVIEW
- Data Acquisition in LabVIEW
- Control and Simulation in LabVIEW
- LabVIEW MathScript
- Linear Algebra in LabVIEW
- Datalogging and Supervisory Control in LabVIEW
- Wireless Data Acquisition in LabVIEW
- Intermediate Topics in LabVIEW
- Advanced Topics in LabVIEW

These Tutorials are available for download from my blog:  http://home.hit.no/~hansha

10.3  Additional Resources

www.ni.com

10.4  Examples

In the NI example Finder (Help→Find Examples...) you find lots of useful examples that you can play with or use as a start when creating your own applications.
10.5 Documentation

As part of the LabVIEW installation there is lots of useful documentation.

Tutorial: Introduction to State-based Applications in LabVIEW
# Quick Reference

## LabVIEW Keyboard Shortcuts

### Objects and Movement
- **Shift-click**: Selects multiple objects; adds object to current selection.
- **Shift-↑ (arrow keys)**: Moves selected objects one pixel at a time.
- **Shift-↓ (arrow keys)**: Moves selected objects several pixels at a time.
- **Shift-click (drag)**: Moves objects in one area.
- **Ctrl-click (drag)**: Duplicates selected objects.
- **Ctrl-Shift-click (drag)**: Duplicates selected objects and moves them in one area.
- **Shift-resize**: Resizes object while maintaining aspect ratio.
- **Ctrl-resize**: Resizes object while maintaining center point and aspect ratio.
- **Ctrl-Shift-resize**: Resizes selected object while maintaining center point and aspect ratio.
- **Ctrl-drag a rectangle in open space**: Adds more working space to the front panel or block diagram.
- **Ctrl-A**: Selects all front panel or block diagram items.
- **Ctrl-Shift-A**: Performs last alignment operation on objects.
- **Ctrl-D**: Performs last distribution operation on objects.
- **Double-click open space**: Adds a free label to the front panel or block diagram if automatic tool selection is enabled.
- **Ctrl-mouse wheel**: Scrolls through subdiagrams of a Cont, Index, or Stacked SubVI structure.
- **Spacebar (drag)**: Disables preset alignment position when moving labels or captions.
- **Ctrl-U**: Rotates all wires and reconnects block diagram objects automatically.

### Debugging
- **Ctrl-I**: Stays into node.
- **Ctrl-→**: Steps over node.
- **Ctrl-←**: Steps out of node.

### Basic Editing
- **Ctrl-Z**: Undoes last action.
- **Ctrl-Shift-Z**: Restores last action.
- **Ctrl-X**: Cuts selected objects.
- **Ctrl-C**: Copies selected objects.
- **Ctrl-V**: Pastes last cut or copied objects.

### Navigating the LabVIEW Environment
- **Ctrl-E**: Displays block diagram or front panel windows.
- **Ctrl-#**: Enables or disables grid alignment.
  - (Mac OS): Press the Command-* keys.
- **Ctrl-/**: Maximizes and restores window.
- **Ctrl-T**: Tiles front panel and block diagram windows.
- **Ctrl-F**: Finds objects or text.
- **Ctrl-G**: Searches VIs for most instance of object or text.
- **Ctrl-Shift-G**: Searches VIs for previous instance of object or text.
- **Ctrl-Shift-F**: Displays the Search Results window.
- **Ctrl-Tab**: Cycles through LabVIEW windows.
- **Ctrl-Shift-Tab**: Cycles through LabVIEW windows in reverse order.
- **Ctrl-Shift-N**: Displays the Navigation window.
- **Ctrl-I**: Displays the VI Properties dialog box.
- **Ctrl-L**: Displays the Error list window.
- **Ctrl-Y**: Displays the History window.
- **Ctrl-Shift-W**: Displays the All Windows dialog box.
- **Ctrl-Space**: Displays the Quick Drop dialog box.
  - (Mac OS): Press the Command-Shift-Space key.

### Navigating the VI Hierarchy Window
- **Ctrl-D**: Closes the window.
- **Ctrl-A**: Shows all VIs in the window.
- **Ctrl-click VI**: Displays the subVI and other nodes that make up the VI you select in the window.
- **Enter**: Finds next node that matches the search string.
- **Shift Enter**: Finds previous node that matches the search string.
  - (After initiating a search by typing in the VI Hierarchy window.)

### File Operations
- **Ctrl-N**: Creates a new blank VI.
- **Ctrl-O**: Opens an existing VI.
- **Ctrl-W**: Closes the VI.
- **Ctrl-S**: Saves the VI.
- **Ctrl-Shift-S**: Saves all open files.
- **Ctrl-P**: Prints the window.
- **Ctrl-Q**: Quits LabVIEW.

### Help
- **Ctrl-H**: Displays the Context Help window.
  - (Mac OS): Press the Command-Shift-H keys.
- **Ctrl-Shift-L**: Locks the Context Help window.
- **Ctrl-? or F1**: Displays the LabVIEW Help.

Refer to the LabVIEW Help for keyboard shortcut variations on other system locales and keyboard layouts.
Quick Reference

**Tools and Palettes**

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl</td>
<td>Switches to next most useful tool.</td>
</tr>
<tr>
<td>Shift</td>
<td>Switches to Positioning tool.</td>
</tr>
<tr>
<td>Ctrl-Shift over open space</td>
<td>Switches to Scrolling tool.</td>
</tr>
<tr>
<td>Spacebar</td>
<td>Toggles between two most common tools.</td>
</tr>
<tr>
<td>Shift-Tab</td>
<td>Enables automatic tool selection.</td>
</tr>
<tr>
<td>Tab</td>
<td>Orders through four most common tools if you disabled automatic tool selection by clicking the Automatic Tool Selection button. Otherwise, enables automatic tool selection.</td>
</tr>
<tr>
<td>Fp→</td>
<td>Navigates temporary Controls and Functions palettes.</td>
</tr>
<tr>
<td>Fp→</td>
<td>Navigates into a temporary palette.</td>
</tr>
<tr>
<td>Esc</td>
<td>Navigates out of a temporary palette.</td>
</tr>
<tr>
<td>Shift-right-click</td>
<td>Displays a temporary version of the Tools palette at the location of the cursor.</td>
</tr>
</tbody>
</table>

*1 If automatic tool selection is disabled.*

**SubVIs**

<table>
<thead>
<tr>
<th>Shortcut or Command</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-click subVI</td>
<td>Displays subVI front panel.</td>
</tr>
<tr>
<td>Ctrl-double-click subVI</td>
<td>Displays subVI block diagram and front panel.</td>
</tr>
<tr>
<td>Drag Vi icon to block diagram</td>
<td>Places that Vi as a subVI on the block diagram.</td>
</tr>
<tr>
<td>Shift-drag Vi icon to block diagram</td>
<td>Places that Vi as a subVI on the block diagram with constants wired for controls that have non-default values.</td>
</tr>
<tr>
<td>Ctrl-right-click block diagram and select Vi from palette</td>
<td>Opens the front panel of that VI.</td>
</tr>
</tbody>
</table>

**Wiring**

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-E</td>
<td>Removes all broken wires.</td>
</tr>
<tr>
<td>Esc, right-click, or click terminal</td>
<td>While wiring, cancels a wire you started.</td>
</tr>
<tr>
<td>Single-click wire</td>
<td>Selects one segment.</td>
</tr>
<tr>
<td>Double-click wire</td>
<td>Selects a branch.</td>
</tr>
<tr>
<td>Triple-click wire</td>
<td>Selects entire wire.</td>
</tr>
<tr>
<td>A</td>
<td>While wiring, disables automatic wire routing temporarily.</td>
</tr>
<tr>
<td>Double-click</td>
<td>While wiring, drops down wire without connecting it.</td>
</tr>
<tr>
<td>spacer</td>
<td>While wiring, switches the direction of a wire between horizontal and vertical.</td>
</tr>
<tr>
<td>spacebar</td>
<td>While moving objects, toggles automatic wiring.</td>
</tr>
<tr>
<td>Ctrl-click input on function with two inputs</td>
<td>Switches the two input wires.</td>
</tr>
<tr>
<td>Shift-click</td>
<td>While wiring, undoes last point where you set a wire.</td>
</tr>
</tbody>
</table>

**Execution**

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-B</td>
<td>Runs the VI.</td>
</tr>
<tr>
<td>Ctrl-C</td>
<td>Stops the VI.</td>
</tr>
<tr>
<td>Ctrl-M</td>
<td>Changes to run or edit mode.</td>
</tr>
<tr>
<td>Ctrl-Run button</td>
<td>Recompiles the current VI.</td>
</tr>
<tr>
<td>Ctrl-Shift Run button</td>
<td>Recompiles all VIs in memory.</td>
</tr>
<tr>
<td>Ctrl-1</td>
<td>Moves key focus inside an array or cluster.</td>
</tr>
<tr>
<td>Ctrl-1</td>
<td>Moves key focus outside an array or cluster.</td>
</tr>
<tr>
<td>Tab</td>
<td>Navigates the controls or indicators according to tabbing order.</td>
</tr>
<tr>
<td>Shift-Tab</td>
<td>Navigates backward through the controls or indicators.</td>
</tr>
</tbody>
</table>

*1 While the VI is running.*

Note: The Ctrl key in these shortcuts corresponds to the Option or Command key on Mac OS and the Alt key on Linux.
Hans-Petter Halvorsen, M.Sc.

E-mail: hans.p.halvorsen@hit.no

Blog: http://home.hit.no/~hansha/

University College of Southeast Norway

www.usn.no