State Notation Language and the Sequencer

Andrew Johnson
APS Engineering Support Division

USPAS June 2010 — EPICS Control Systems
Outline

- What is State Notation Language (SNL)
- Where it fits in the EPICS toolkit
- Components of a state notation program
- Some notes on the Sequencer runtime
- Building, running and debugging a state notation program
- Additional features
- When to use it

- This talk does not cover all the features of the sequencer
- Consult the reference manual for more information
  - http://www-csr.bessy.de/control/SoftDist/sequencer/
SNL and the Sequencer

- The sequencer runs programs written in State Notation Language (SNL)
- SNL is a ‘C’ like language to facilitate programming of sequential operations
- Fast execution using compiled code
- Programming interface to extend EPICS in the real-time environment
- Common uses
  - Automated start-up and sequencing for subsystems like vacuum or RF where coordination of multiple components needed
  - Provide fault recovery or transition to a safe state
  - Provide automatic calibration of equipment
Where is the Sequencer?

LAN

IOC

Channel Access

Database

Sequencer

Device Support

I/O Hardware
Where is the Sequencer?
The Best Place for the Sequencer

- Traditionally sequencers run in the IOC
- Recent versions can be run either within an IOC or as a standalone program on a workstation
- Locating them within the IOC they control makes them easier to manage and independent of network issues
- Running them on a workstation can make testing and debugging easier
- On a workstation, SNL provides an easy way to write simple CA client programs
Some Definitions

- **SNL**: State Notation Language
- **SNC**: State Notation Compiler
- **Sequencer**: The tool that executes the compiled SNL code
- **Program**: A complete SNL application, consisting of declarations and one or more state sets
- **State Set**: A set of states that make a complete finite state machine
- **State**: A particular mode of the state set in which it remains until one of its transition conditions evaluates to TRUE
SNL implements State Transition Diagrams

State A

Transition A to B

Event

Action

State B
Example State Transition Diagram

Start

Low vacuum

- pressure < 4.9 µTorr
  - open valve, turn on cryo pump

High vacuum

- pressure > 5.1 µTorr
  - close valve, turn off cryo pump
SNL: General Structure and Syntax

```plaintext
program  program_name
declarations

ss  state_set_name {  
    state  state_name {  
        entry {  
            entry action statements  
        }  
        when (event) {  
            action statements  
        }  
        state  next_state_name  
        when (event) {  
            ...  
        }  
        state  next_state_name  
        exit{  
            exit action statements  
        }  
    }  
    state  state_name {  
        ...  
    }  
}
```

SNL: General Structure and Syntax

**program name**

A program may contain multiple state sets. The program *name* is used as a handle to the sequencer manager for state programs.

**ss name {**
**state name {**

Each state set becomes a separate task or thread.

A state is somewhere the task waits for events. When an event occurs it checks to see which action it should execute. The first state defined in a state set is the initial state.

**option flag;**

A state-specific option.

**when (event) {**

Define events for which this state waits.

**} state next**

Specifies the state to go to when these actions are complete.

**entry {actions}**

Actions to do on entering this state. With **option -e**; it will do these actions even if it enters from the same state.

**exit {actions}**

Actions to do on exiting this state. With **option -x**; it will do these actions even if it exits to the same state.
Declarations - Variables

- Appear before a state set and have a scope of the entire program.
- Scalar variables
  - `int` `var_name`;
  - `short` `var_name`;
  - `long` `var_name`;
  - `char` `var_name`;
  - `float` `var_name`;
  - `double` `var_name`;
  - `string` `var_name`; /* 40 characters */
- Array variables: 1 or 2 dimensions, no strings
  - `int` `var_name[num_elements];`
  - `short` `var_name[num_elements];`
  - `long` `var_name[num_elements];`
  - `char` `var_name[num_elements];`
  - `float` `var_name[num_elements];`
  - `double` `var_name[num_elements];`
Declarations - Assignments

- Assignment connects a variable to a channel access PV name
  ```
  float pressure;
  assign pressure to "CouplerPressureRB1";
  double pressures[3];
  assign pressures to {"CouplerPressureRB1",
  "CouplerPressureRB2", "CouplerPressureRB3"};
  ```

- To use these channels in when clauses, they must be monitored
  ```
  monitor pressure;
  monitor pressures;
  ```

- Use preprocessor macros to aid readability:
  ```
  #define varMon(t,n,c) t n; assign n to c; monitor n;
  varMon(float, pressure, "PressureRB1")
  ```
Declarations - Event Flags

- Event flags are used to communicate events between state sets, or to receive explicit event notifications from Channel Access
- Declare them like this:
  ```
  evflag  event_flag_name;
  ```
- An event flag can be synchronized with a monitored variable
  ```
  sync  var_name  event_flag_name;
  ```
- The flag will then be set when a monitor notification arrives, e.g.
  ```
  evflag  pressure_event;
  sync  pressure  pressure_event;
  ```
Events

Event: A specific condition on which associated actions are run and a state transition is made.
Possible events:

- Change in value of a variable that is being monitored:
  \texttt{when (achan < 10.0)}

- A timer event (this is not a task delay!):
  \texttt{when (delay(1.5))}
  - The delay time is in seconds and is a double; literal constant arguments to the delay function \textit{must} contain a decimal point.
  - The timer starts when the state containing it was entered.
  - Use the state specific \texttt{option -t}; to stop it from being reset when transitioning to the same state.
Events (continued)

- The state of an event flag:
  
  ```
  when (efTestAndClear(myflag))
  when (efTest(myflag))
  ```
  
  - `efTest()` does not clear the flag. `efClear()` must be called sometime later to avoid an infinite loop.
  - If the flag is synced to a monitored variable, it will be set when the channel sends a value update.
  - The event flag can also be set by any state set in the program using `efSet(event_flag_name)`

- Any change in the channel access connection status:
  
  ```
  when (pvConnectCount() < pvChannelCount())
  when (pvConnected(mychan))
  ```

- Any combination of the above event types
Action Statements

- **Built-in action function, e.g.** :
  - `pvPut(var_name);`
  - `pvGet(var_name);`
  - `efSet(event_flag_name);`
  - `efClear(event_flag_name);`

- **Almost any valid C statement**
  - `switch()` *is not implemented and code using it must be escaped.*
  - `%%` escapes one line of C code
  - `%{`
    escape any number of lines of C code
  }%
Example - State Definitions and Transitions

Initial State

- pressure > .0000051
  - RoughPump on
  - CryoPump off
  - Valve closed

Low Vacuum

- pressure <= .0000049
  - RoughPump off
  - CryoPump on
  - Valve open

High Vacuum

- pressure > .0000051
  - RoughPump on
  - CryoPump off
  - Valve closed

Fault

- 10 minutes
  - RoughPump off
  - CryoPump off
  - Valve closed
Example - Declarations

double pressure;
assign pressure to "Tank1Coupler1PressureRB";
monitor pressure;

short RoughPump;
assign RoughPump to "Tank1Coupler1RoughPump";
short CryoPump;
assign CryoPump to "Tank1Coupler1CryoPump";
short Valve;
assign Valve to "Tank1Coupler1IsolationValve";
string CurrentState;
assign CurrentState to "Tank1Coupler1VacuumState";
Example - State Transitions, Actions Omitted

program vacuum_control

ss coupler_control
{
  state init{
    when (pressure > .0000049){
      } state low_vacuum
    when (pressure <= .0000049){
      } state high_vacuum
  }
  state high_vacuum{
    when (pressure > .0000051){
      } state low_vacuum
  }
  state low_vacuum{
    when (pressure <= .0000049){
      } state high_vacuum
    when (delay(600.0)){
      } state fault
  }
  state fault { 
  
  }
}
Example - Initial State

```c
state init {
    entry {
        strcpy(CurrentState,"Init");
        pvPut(CurrentState);
    }
    when (pressure > .0000049){
        RoughPump = 1;
        pvPut(RoughPump);
        CryoPump = 0;
        pvPut(CryoPump);
        Valve = 0;
        pvPut(Valve);
    }
    state low_vacuum
    when (pressure <= .0000049){
        RoughPump = 0;
        pvPut(RoughPump);
        CryoPump = 1;
        pvPut(CryoPump);
        Valve = 1;
        pvPut(Valve);
    }
    state high_vacuum
}
```
Example - State low_vacuum

```c
state low_vacuum{
    entry {
        strcpy(CurrentState,"Low Vacuum");
        pvPut(CurrentState);
    }
    when (pressure <= .0000049){
        RoughPump = 0;
        pvPut(RoughPump);
        CryoPump = 1;
        pvPut(CryoPump);
        Valve = 1;
        pvPut(Valve);
    }
    state high_vacuum
    when (delay(600.0)) {
        state fault
    }
}
```
Example - State high_vacuum

```c
state high_vacuum{
    entry {
        strcpy(CurrentState,"High Vacuum");
        pvPut(CurrentState);
    }
    when (pressure > .0000051){
        RoughPump = 1;
        pvPut(RoughPump);
        CryoPump = 0;
        pvPut(CryoPump);
        Valve = 0;
        pvPut(Valve);
    } state low_vacuum
}
```
**Example - State fault**

```plaintext
state fault{
    entry{
        strcpy(CurrentState,"Vacuum Fault");
        pvPut(CurrentState);
    }
}
```
Building an SNL program

- Use editor to build the source file. File name must end with "*.st" or "*.stt", e.g. "example.st"
- "make" automates these steps:
  - Runs the C preprocessor on "*.st" files, but not on "*.stt" files.
  - Compiles the state program with SNC to produce C code:
    - snc example.st -> example.c
  - Compiles the resulting C code with the C compiler:
    - cc example.c -> example.o
  - The object file example.o becomes part of the application library, ready to be linked into an IOC binary.
  - The executable file "example" can be created instead.
Run Time Sequencer

- The sequencer executes the state program
- It is implemented as an event-driven application; no polling is needed
- Each state set becomes an operating system thread
- The sequencer manages connections to database channels through Channel Access
- It provides support for channel access get, put, and monitor operations
- It supports asynchronous execution of delays, event flag, pv put and pv get functions
- Only one copy of the sequencer code is required to run multiple programs
- Commands are provided to display information about the state programs currently executing
Executing a State Program

From an IOC console

- On vxWorks:
  seq &vacuum_control

- On other operating systems:
  seq vacuum_control

- To stop the program
  seqStop "vacuum_control"
Debugging

- Use the sequencer's query commands:
  
  \texttt{seqShow} \\
  \textit{displays information on all running state programs} \\
  \texttt{seqShow\ vacuum\_control} \\
  \textit{displays detailed information on program} \\
  \texttt{seqChanShow\ vacuum\_control} \\
  \textit{displays information on all channels} \\
  \texttt{seqChanShow\ vacuum\_control, "-"} \\
  \textit{displays information on all disconnected channels} \\
  \texttt{seqcar} \\
  \textit{displays information on all channel access channels}
Debugging (continued)

- Use printf functions to print to the console
  
  ```c
  printf("Here I am in state xyz \n");
  ```

- Put strings to pvs
  
  ```c
  sprintf(seqMsg1, "Here I am in state xyz");
  pvPut(seqMsg1);
  ```

- On vxWorks you can reload and restart
  
  ```
  seqStop vacuum_control
  ... edit, recompile ...
  ld < example.o
  seq &vacuum_control
  ```
Additional Features

- **Connection management:**
  
  ```
  when (pvConnectCount() != pvChannelCount())
  when (pvConnected(Vin))
  ```

- **Macros:**
  
  ```
  assign Vout to "\{unit\}:OutputV";
  ```

  - must use the +r compiler options for this if more than one copy of the sequence is running on the same ioc

  ```
  seq &example, "unit=HV01"
  ```

- **Some common SNC program options:**
  
  - +r make program reentrant (default is -r)
  - -c don't wait for all channel connections (default is +c)
  - +a asynchronous pvGet() (default is -a)
  - -w don't print compiler warnings (default is +w)
Additional Features (continued)

- Access to channel alarm status and severity:
  \[ \text{pvStatus}(\text{var\_name}) \]
  \[ \text{pvSeverity}(\text{var\_name}) \]

- Queued monitors save CA monitor events in a queue in the order they come in, rather than discarding older values when the program is busy
  \[ \text{syncQ} \text{ var\_name to event\_flag\_name [queue\_length]} \]
  \[ \text{pvGetQ}(\text{var\_name}) \]
  - removes oldest value from variable's monitor queue. Remains true until queue is empty.
  \[ \text{pvFreeQ}(\text{var\_name}) \]
Advantages of SNL

- Can implement complicated algorithms
- Can stop, reload, restart a sequence program without rebooting
- Interact with the operator through string records and mbbo records
- C code can be embedded as part of the sequence
- All Channel Access details are taken care of for you
- File access can be implemented as part of the sequence
When to use the sequencer

- For sequencing complex events
- E.g. parking and unparking a telescope mirror
Should I Use the Sequencer?

START

CAN I DO THIS IN A DB?

CAN I DO THIS IN A DB?

USE THE SEQUENCER

USE A DATABASE

END
Acknowledgments

- Slides for this presentation have been taken from talks prepared by the following people
  - Bob Dalesio (LANL/SNS/LCLS)
  - Deb Kerstiens (LANL)
  - Rozelle Wright (LANL)
  - Ned Arnold (Argonne)
  - John Maclean (Argonne)