Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

EPICS State Notation Language and Sequencer

Version 2.1

Ben Franksen

Helmholtz-Zentrum Berlin für Materialien und Energie (HZB)
(Wilhelm-Conrad-Röntgen Campus / BESSY II)

EPICS Meeting 2011 @ PSI, 2011

Prologue: A Small Glossary

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

SNL State Notation Language

PV Process Variable

CA Channel Access

Sequencer The SNL runtime library

Sequencer The project that implements SNL

Prologue: A Small Glossary

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

SNL	State Notation Language
PV	Process Variable
CA	Channel Access
Sequencer	The SNL runtime library
Sequencer	The project that implements SNL

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

Prologue: A Small Glossary

SNL State Notation Language

PV Process Variable

CA Channel Access

Sequencer The SNL runtime library

Sequencer The project that implements SNL

Prologue: A Small Glossary

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

SNL	State Notation Language
PV	Process Variable
CA	Channel Access
Sequencer	The SNL runtime library
Sequencer	The project that implements SNL

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

Prologue: A Small Glossary

SNL	State Notation Language
PV	Process Variable
CA	Channel Access
Sequencer	The SNL runtime library
Sequencer	The project that implements SNL

Outline

1. Introduction
2. What's New
3. Next Steps
4. Summary

Outline

1. Introduction
2. What's New
3. Next Steps
4. Summary

Outline

1. Introduction
2. What's New
3. Next Steps
4. Summary

Outline

1. Introduction
2. What's New
3. Next Steps
4. Summary

Outline

1. Introduction
   - SNL in a Nutshell
   - Limitations
2. What's New
   - Overview
   - Scoped Variables
   - Safe Mode
3. Next Steps
4. Summary

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

What is SNL?

- a *domain specific language*
- for programming control applications
- in co-operation with an EPICS database
- typically used when control flow gets more complex than can be easily achieved with records and links
- a (small) subset of the C language, plus features for
  - specifying *finite state machines* (state sets, keyword `ss`)
  - binding program variables to PVs (`assign`)
  - specifying that such variables are to be automatically updated whenever the PV changes (`monitor`)
  - explicit interaction with PVs (`pvPut`, `pvGet`)
- compiler translates SNL to C
- internally PVs are accessed via Channel Access

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

skip animation

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Example

```
program level_check

float v;
assign v to "Input_voltage";
monitor v;

short light;
assign light to "Indicator_light";

ss volt_check {
  state light_off {
    when (v > 5.0) {
      /* turn light on */
      light = 1;
      pvPut(light);
    } state light_on
  }

  state light_on {
    when (v < 5.0) {
      /* turn light off */
      light = 0;
      pvPut(light);
    } state light_off
  }
}
```

- declare variables
- connect them to EPICS PV names
- automatically update v to the PV's value
- define a state set
- if in this state
- ... and this condition holds
- ... then execute these actions
- ... and switch to new state

Outline

1. Introduction
   - SNL in a Nutshell
   - Limitations
2. What's New
   - Overview
   - Scoped Variables
   - Safe Mode
3. Next Steps
4. Summary

Traditional Limitations of SNL

- No type definitions
- No procedures or other abstraction facilities
- Only global variables, no initialization
- Restricted set of data types, roughly corresponding to EPICS field types:
  - `char`, `short`, `int`, `long` (and `unsigned` versions)
  - `float`, `double`
  - `string` (synonym for `char[MAX_STRING_SIZE]`)
  - one or two dimensional arrays of above
- Escape to C code often used as work-around

Traditional Limitations of SNL

- No type definitions
- No procedures or other abstraction facilities
- Only global variables, no initialization
- Restricted set of data types, roughly corresponding to EPICS field types:
  - `char`, `short`, `int`, `long` (and `unsigned` versions)
  - `float`, `double`
  - `string` (synonym for `char[MAX_STRING_SIZE]`)
  - one or two dimensional arrays of above
- Escape to C code often used as work-around

Traditional Limitations of SNL

- No type definitions
- No procedures or other abstraction facilities
- Only global variables, no initialization
- Restricted set of data types, roughly corresponding to EPICS field types:
  - `char`, `short`, `int`, `long` (and `unsigned` versions)
  - `float`, `double`
  - `string` (synonym for `char[MAX_STRING_SIZE]`)
  - one or two dimensional arrays of above
- Escape to C code often used as work-around

Traditional Limitations of SNL

- No type definitions
- No procedures or other abstraction facilities
- Only global variables, no initialization
- Restricted set of data types, roughly corresponding to EPICS field types:
  - `char`, `short`, `int`, `long` (and `unsigned` versions)
  - `float`, `double`
  - `string` (synonym for `char[MAX_STRING_SIZE]`)
  - one or two dimensional arrays of above
- Escape to C code often used as work-around

Traditional Limitations of SNL

- No type definitions
- No procedures or other abstraction facilities
- Only global variables, no initialization
- Restricted set of data types, roughly corresponding to EPICS field types:
  - `char`, `short`, `int`, `long` (and `unsigned` versions)
  - `float`, `double`
  - `string` (synonym for `char[MAX_STRING_SIZE]`)
  - one or two dimensional arrays of above
- Escape to C code often used as work-around

Traditional Limitations of SNL

- No type definitions
- No procedures or other abstraction facilities
- Only global variables, no initialization
- Restricted set of data types, roughly corresponding to EPICS field types:
  - `char`, `short`, `int`, `long` (and `unsigned` versions)
  - `float`, `double`
  - `string` (synonym for `char[MAX_STRING_SIZE]`)
  - one or two dimensional arrays of above
- Escape to C code often used as work-around

Outline

1. Introduction
   - SNL in a Nutshell
   - Limitations
2. What's New
   - Overview
   - Scoped Variables
   - Safe Mode
3. Next Steps
4. Summary

What's new in 2.1?

- Syntactic restrictions lifted
  - allow local variables in all blocks
  - support more variable types
  - declarations with multiple variables and initialization
- Additions: you can now
  - jump to a new state from action code: `state new_state;`
  - exit program instead of transition to a new state: `when(...) {...} exit`
  - avoid "used but not defined" warnings by declaring `foreign` identifiers (variables, macros, whatever)
- Safe Mode: avoid race conditions for global variables
- Improved documentation
- Started adding automated regression tests (no complete coverage yet)
- Many bugs fixed, general code cleanup
- Deep refactorings may have introduced new bugs (but hopefully only shallow ones)

What's new in 2.1?

- Syntactic restrictions lifted
  - allow local variables in all blocks
  - support more variable types
  - declarations with multiple variables and initialization
- Additions: you can now
  - jump to a new state from action code: `state new_state;`
  - exit program instead of transition to a new state: `when(...) {...} exit`
  - avoid "used but not defined" warnings by declaring `foreign` identifiers (variables, macros, whatever)
- Safe Mode: avoid race conditions for global variables
- Improved documentation
- Started adding automated regression tests (no complete coverage yet)
- Many bugs fixed, general code cleanup
- Deep refactorings may have introduced new bugs (but hopefully only shallow ones)

What's new in 2.1?

- Syntactic restrictions lifted
  - allow local variables in all blocks
  - support more variable types
  - declarations with multiple variables and initialization
- Additions: you can now
  - jump to a new state from action code: `state new_state;`
  - exit program instead of transition to a new state: `when(...) {...} exit`
  - avoid "used but not defined" warnings by declaring `foreign` identifiers (variables, macros, whatever)
- Safe Mode™: avoid race conditions for global variables
- Improved documentation
- Started adding automated regression tests (no complete coverage yet)
- Many bugs fixed, general code cleanup
- Deep refactorings may have introduced new bugs (but hopefully only shallow ones)

What's new in 2.1?

- Syntactic restrictions lifted
  - allow local variables in all blocks
  - support more variable types
  - declarations with multiple variables and initialization
- Additions: you can now
  - jump to a new state from action code: `state new_state;`
  - exit program instead of transition to a new state: `when(...) {...} exit`
  - avoid "used but not defined" warnings by declaring `foreign` identifiers (variables, macros, whatever)
- Safe Mode: avoid race conditions for global variables
- Improved documentation
- Started adding automated regression tests (no complete coverage yet)
- Many bugs fixed, general code cleanup
- Deep refactorings may have introduced new bugs (but hopefully only shallow ones)

What's new in 2.1?

- Syntactic restrictions lifted
  - allow local variables in all blocks
  - support more variable types
  - declarations with multiple variables and initialization
- Additions: you can now
  - jump to a new state from action code: `state new_state;`
  - exit program instead of transition to a new state: `when(...) {...} exit`
  - avoid "used but not defined" warnings by declaring `foreign` identifiers (variables, macros, whatever)
- Safe Mode: avoid race conditions for global variables
- Improved documentation
- Started adding automated regression tests (no complete coverage yet)
- Many bugs fixed, general code cleanup
- Deep refactorings may have introduced new bugs (but hopefully only shallow ones)

What's new in 2.1?

- Syntactic restrictions lifted
  - allow local variables in all blocks
  - support more variable types
  - declarations with multiple variables and initialization
- Additions: you can now
  - jump to a new state from action code: `state new_state;`
  - exit program instead of transition to a new state: `when(...) {...} exit`
  - avoid "used but not defined" warnings by declaring `foreign` identifiers (variables, macros, whatever)
- Safe Mode: avoid race conditions for global variables
- Improved documentation
- Started adding automated regression tests (no complete coverage yet)
- Many bugs fixed, general code cleanup
- Deep refactorings may have introduced new bugs (but hopefully only shallow ones)

What's new in 2.1?

- Syntactic restrictions lifted
  - allow local variables in all blocks
  - support more variable types
  - declarations with multiple variables and initialization
- Additions: you can now
  - jump to a new state from action code: `state new_state;`
  - exit program instead of transition to a new state: `when(...) {...} exit`
  - avoid "used but not defined" warnings by declaring `foreign` identifiers (variables, macros, whatever)
- Safe Mode: avoid race conditions for global variables
- Improved documentation
- Started adding automated regression tests (no complete coverage yet)
- Many bugs fixed, general code cleanup
- Deep refactorings may have introduced new bugs (but hopefully only shallow ones)

Outline

1. Introduction
   - SNL in a Nutshell
   - Limitations
2. What's New
   - Overview
   - Scoped Variables
   - Safe Mode
3. Next Steps
4. Summary

Scoped Variables

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - behaves like built-in variables in an action block
  - at the start of states and state set blocks
    - possible to define local variables, accessible only in the state or state set
    - but needn't all be at the top of the block
    - interleaved with other statements, e.g. assignments, sleep
    - can be used in all parts of the state or state set

Scoped Variables

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assigned`, `monitored`, etc. like globals

Scoped Variables

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assigned`, `monitored`, etc. like globals

Scoped Variables

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assigned`, `monitored`, etc. like globals

Scoped Variables

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assign`ed, `monitor`ed, etc. like globals

Scoped Variables

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assign`ed, `monitor`ed, etc. like globals

Scoped Variables

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assign`ed, `monitor`ed, etc. like globals

Scoped Variables

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assign`ed, `monitor`ed, etc. like globals

Scoped Variables

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assign`ed, `monitor`ed, etc. like globals

Scoped Variables

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Variables can be declared on all levels
- They come in two variants
  - nested anywhere inside action blocks (state transition, entry, exit)
    - temporary, like local (`auto`) variables in C
    - no `assign` to PVs allowed
  - at the start of states and state set blocks
    - persistent until program ends, like C static variables
    - but reentrant (if `option +r` is active)
    - and statically scoped, i.e. invisible to code outside the block
    - *can* be `assign`ed, `monitor`ed, etc. like globals

Outline

1. Introduction
   - SNL in a Nutshell
   - Limitations
2. What's New
   - Overview
   - Scoped Variables
   - Safe Mode
3. Next Steps
4. Summary

Motivation

From the SNL mission statement:

> *The state notation language allows programming sequential operations that interact with EPICS process variables without the usual complexity involved with task scheduling, semaphores, event handling, and I/O programming. [...]*

Motivation

From the SNL mission statement:

> *The state notation language allows programming sequential operations that interact with EPICS process variables without the usual complexity involved with task scheduling, semaphores, event handling, and I/O programming. [...]*

Motivation

From the SNL mission statement:

> *The state notation language allows programming sequential operations that interact with EPICS process variables without the usual complexity involved with task scheduling, semaphores, event handling, and I/O programming. [...]*

Unfortunately this is not the case.

The Problem

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

The Problem

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

The Problem

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

The Problem

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

The Problem

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

The Problem

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

The Problem

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

The Problem

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

The Problem

- SNL variables are not protected from concurrent access
- Threads involved
  - one or more state sets that access a variable
  - the CA callback thread, performing the variable updates (monitor, get completion)
- Variables can become corrupted if access is non-atomic
- Resulting failures are extremely difficult to debug
- There is no reliable work-around for the SNL programmer
  - CA updates can intervene at any time
- A serious bug that *has* to be fixed
- But how?

Solution

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Solution

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Solution

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Solution

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Solution

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Solution

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Solution

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Solution

- Each state set operates on its own copy of each variable
- Aynchronous CA updates operate on yet another copy
- State sets are completely isolated from each other and from asynchronous variable updates
- Synchronization (exchange of data) happens only outside user visible (SNL) code:
  - for monitored variables: immediately before `when`-conditions are checked
  - for all assigned variables: inside `pvGet` / `pvPut` and related built-in functions (but only the synchronous versions that actually access the variable)
- Nice side effect: conditions established in `when(...)` clauses are never invalidated except explicitly by the invoked action code
- Not completely backwards compatible, therefore must be enabled with an `option +s`. Implies re-entrant mode.

Anonymous Pvs

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs™
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag ≡ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Anonymous Pvs

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Safe mode prevents state sets from communicating via global variables
- Event flags work but can communicate only booleans
- For other data types: use Anonymous PVs
- Syntax: global variable assigned to `""`, short form:
  - `assign x;` (scalars)
  - `assign x {};` (arrays)
- Use `pvPut`, `pvGet`, and `monitor` like with normal (named) PVs
- Named and anononymous PVs can be freely exchanged, all built-in functions behave in the the same way
- event flag $\equiv$ anonymous boolean PV (modulo `sync`)

Outline

1. Introduction
   - SNL in a Nutshell
   - Limitations
2. What's New
   - Overview
   - Scoped Variables
   - Safe Mode
3. Next Steps
4. Summary

Next Steps

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Integrate code from DESY to support redundant IOCs
- Add more regression tests
- Remove stuff that is ugly and obsolete
  - PV layer (a C++ wrapper around CA or other protocols)
  - devSequencer (device support for monitoring seq's internal state)
- Lift further restrictions
  - allow foreign (C) types to be used in declarations
  - define data types (e.g. `struct`) in SNL?
- New notation for `assign`, `monitor`, `sync`, etc.
  - all extra attributes of a variable in one place
  - obviate the need for user defined CPP macros
  - need a good idea for the syntax
- Add some kind of abstraction facility
  - parameterizable state sets (a.k.a. procedures)

Next Steps

- Integrate code from DESY to support redundant IOCs
- Add more regression tests
- Remove stuff that is ugly and obsolete
  - PV layer (a C++ wrapper around CA or other protocols)
  - devSequencer (device support for monitoring seq's internal state)
- Lift further restrictions
  - allow foreign (C) types to be used in declarations
  - define data types (e.g. `struct`) in SNL?
- New notation for `assign`, `monitor`, `sync`, etc.
  - all extra attributes of a variable in one place
  - obviate the need for user defined CPP macros
  - need a good idea for the syntax
- Add some kind of abstraction facility
  - parameterizable state sets (a.k.a. procedures)

Next Steps

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Integrate code from DESY to support redundant IOCs
- Add more regression tests
- Remove stuff that is ugly and obsolete
  - PV layer (a C++ wrapper around CA or other protocols)
  - devSequencer (device support for monitoring seq's internal state)
- Lift further restrictions
  - allow foreign (C) types to be used in declarations
  - define data types (e.g. `struct`) in SNL?
- New notation for `assign`, `monitor`, `sync`, etc.
  - all extra attributes of a variable in one place
  - obviate the need for user defined CPP macros
  - need a good idea for the syntax
- Add some kind of abstraction facility
  - parameterizable state sets (a.k.a. procedures)

Next Steps

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Integrate code from DESY to support redundant IOCs
- Add more regression tests
- Remove stuff that is ugly and obsolete
  - PV layer (a C++ wrapper around CA or other protocols)
  - devSequencer (device support for monitoring seq's internal state)
- Lift further restrictions
  - allow foreign (C) types to be used in declarations
  - define data types (e.g. `struct`) in SNL?
- New notation for `assign`, `monitor`, `sync`, etc.
  - all extra attributes of a variable in one place
  - obviate the need for user defined CPP macros
  - need a good idea for the syntax
- Add some kind of abstraction facility
  - parameterizable state sets (a.k.a. procedures)

Next Steps

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Integrate code from DESY to support redundant IOCs
- Add more regression tests
- Remove stuff that is ugly and obsolete
  - PV layer (a C++ wrapper around CA or other protocols)
  - devSequencer (device support for monitoring seq's internal state)
- Lift further restrictions
  - allow foreign (C) types to be used in declarations
  - define data types (e.g. `struct`) in SNL?
- New notation for `assign`, `monitor`, `sync`, etc.
  - all extra attributes of a variable in one place
  - obviate the need for user defined CPP macros
  - need a good idea for the syntax
- Add some kind of abstraction facility
  - parameterizable state sets (a.k.a. procedures)

Next Steps

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- Integrate code from DESY to support redundant IOCs
- Add more regression tests
- Remove stuff that is ugly and obsolete
  - PV layer (a C++ wrapper around CA or other protocols)
  - devSequencer (device support for monitoring seq's internal state)
- Lift further restrictions
  - allow foreign (C) types to be used in declarations
  - define data types (e.g. `struct`) in SNL?
- New notation for `assign`, `monitor`, `sync`, etc.
  - all extra attributes of a variable in one place
  - obviate the need for user defined CPP macros
  - need a good idea for the syntax
- Add some kind of abstraction facility
  - parameterizable state sets (a.k.a. procedures) ▸ explain details

Summary

Sequencer 2.1
Ben Franksen
Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- For a long time the Sequencer suffered from neglect. This has changed, it is again actively maintained, new features are being added, bugs are getting fixed, even deep and difficult ones.
- The 2.1.x releases prove that many of the traditional limitations and restrictions in SNL can be overcome with reasonable effort by suitable re-engineering.
  Expect further improvements in version 2.2.

Summary

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

- For a long time the Sequencer suffered from neglect. This has changed, it is again actively maintained, new features are being added, bugs are getting fixed, even deep and difficult ones.
- The 2.1.x releases prove that many of the traditional limitations and restrictions in SNL can be overcome with reasonable effort by suitable re-engineering.
  Expect further improvements in version 2.2.

Further Reading

Introduction
SNL in a Nutshell
Limitations
What's New
Overview
Scoped Variables
Safe Mode
Problem
Solution
Next Steps
Summary

http://www-csr.bessy.de/control/SoftDist/sequencer/

5 Appendix

- Types and Declarations
- Declaration Example
- Rationale for Safe Mode
- Procedures

Outline

5 Appendix
- Types and Declarations
- Declaration Example
- Rationale for Safe Mode
- Procedures

Types and Declarations

- Type expressions
  - any combination of pointer + array + base types (e.g. `string (*a4ps[4])[5]`)
  - fixed size integral types as defined in C99: `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, `uint32_t`
- Declaration Syntax
  - Multiple variables in one declaration
  - Initialization, including aggregate (array) initializers
- All this should work as in C

Types and Declarations

- Type expressions
  - any combination of pointer + array + base types (e.g. `string (*a4ps[4])[5]`)
  - fixed size integral types as defined in C99: `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, `uint32_t`
- Declaration Syntax
  - Multiple variables in one declaration
  - Initialization, including aggregate (array) initializers
- All this should work as in C

Types and Declarations

- Type expressions
  - any combination of pointer + array + base types (e.g. `string (*a4ps[4])[5]`)
  - fixed size integral types as defined in C99: `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, `uint32_t`
- Declaration Syntax
  - Multiple variables in one declaration
  - Initialization, including aggregate (array) initializers
- All this should work as in C

Types and Declarations

- Type expressions
  - any combination of pointer + array + base types (e.g. `string (*a4ps[4])[5]`)
  - fixed size integral types as defined in C99: `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, `uint32_t`
- Declaration Syntax
  - Multiple variables in one declaration
  - Initialization, including aggregate (array) initializers
- All this should work as in C

Types and Declarations

- Type expressions
  - any combination of pointer + array + base types (e.g. `string (*a4ps[4])[5]`)
  - fixed size integral types as defined in C99: `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, `uint32_t`
- Declaration Syntax
  - Multiple variables in one declaration
  - Initialization, including aggregate (array) initializers
- All this should work as in C

Types and Declarations

- Type expressions
  - any combination of pointer + array + base types (e.g. `string (*a4ps[4])[5]`)
  - fixed size integral types as defined in C99: `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, `uint32_t`
- Declaration Syntax
  - Multiple variables in one declaration
  - Initialization, including aggregate (array) initializers
- All this should work as in C

Types and Declarations

- Type expressions
  - any combination of pointer + array + base types (e.g. `string (*a4ps[4])[5]`)
  - fixed size integral types as defined in C99: `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, `uint32_t`
- Declaration Syntax
  - Multiple variables in one declaration
  - Initialization, including aggregate (array) initializers
- All this should work as in C

Outline

5 Appendix
- Types and Declarations
- Declaration Example
- Rationale for Safe Mode
- Procedures

A Real-World Example

```
ss AICresponsibleStates {
  state st_process {
    int resp[MAXSEG][MAXERR] = {
      /* Segment * ERROR: blTs, blW, idTs, idW, d1ts, d1w, d2ts, d2w, landau, vacuum */
      /*****************************************************************************/
      /* seg[1]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[2]  */     {  2,    2,   2,   1,   1,    1,   1,    2,   1,      1},
      /* seg[3]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[4]  */     {  2,    2,   2,   1,   1,    1,   1,    1,   1,      1},
      /* seg[5]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[6]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
    };
    when(delay(1)) {
      int isE = 0, isM = 0;
      int s, e;
      for( s = 0; s < MAXSEG; s++ ) {
        if( seg[s] == 1 ) {        /* segments: 0=ok */
          for( e = 0; e < MAXERR; e++ ) {
            if( err[e] == 0 ) {    /* errors: 1=ok */
              if( resp[s][e] == 1 ) {
                isM = 1;
              }
              if( resp[s][e] == 2 ) {
                isE = 2;
              }
            }
          }
        }
      }
      AICresponsible = isE + isM;
      pvPut(AICresponsible);
    } state st_process
  }
}
```

- declaration local to state
- (multi-variable) declarations local to transition
- scalar initializers
- array initializer

A Real-World Example


```
ss AICresponsibleStates {
  state st_process {
    int resp[MAXSEG][MAXERR] = {
      /* Segment * ERROR: blTs, blW, idTs, idW, d1ts, d1w, d2ts, d2w, landau, vacuum */
      /*****************************************************************************/
      /* seg[1]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[2]  */     {  2,    2,   2,   1,   1,    1,   1,    2,   1,      1},
      /* seg[3]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[4]  */     {  2,    2,   2,   1,   1,    1,   1,    1,   1,      1},
      /* seg[5]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[6]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
    };
    when(delay(1)) {
      int isE = 0, isM = 0;
      int s, e;
      for( s = 0; s < MAXSEG; s++ ) {
        if( seg[s] == 1 ) {        /* segments: 0=ok */
          for( e = 0; e < MAXERR; e++ ) {
            if( err[e] == 0 ) {    /* errors: 1=ok */
              if( resp[s][e] == 1 ) {
                isM = 1;
              }
              if( resp[s][e] == 2 ) {
                isE = 2;
              }
            }
          }
        }
      }
      AICresponsible = isE + isM;
      pvPut(AICresponsible);
    } state st_process
  }
}
```

- declaration local to state
- (multi-variable) declarations local to transition
- scalar initializers
- array initializer

A Real-World Example


```
ss AICresponsibleStates {
  state st_process {
    int resp[MAXSEG][MAXERR] = {
      /* Segment * ERROR: blTs, blW, idTs, idW, d1ts, d1w, d2ts, d2w, landau, vacuum */
      /*****************************************************************************/
      /* seg[1]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[2]  */     {  2,    2,   2,   1,   1,    1,   1,    2,   1,      1},
      /* seg[3]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[4]  */     {  2,    2,   2,   1,   1,    1,   1,    1,   1,      1},
      /* seg[5]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[6]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
    };
    when(delay(1)) {
      int isE = 0, isM = 0;
      int s, e;
      for( s = 0; s < MAXSEG; s++ ) {
        if( seg[s] == 1 ) {        /* segments: 0=ok */
          for( e = 0; e < MAXERR; e++ ) {
            if( err[e] == 0 ) {    /* errors: 1=ok */
              if( resp[s][e] == 1 ) {
                isM = 1;
              }
              if( resp[s][e] == 2 ) {
                isE = 2;
              }
            }
          }
        }
      }
      AICresponsible = isE + isM;
      pvPut(AICresponsible);
    } state st_process
  }
}
```

- declaration local to state
- (multi-variable) declarations local to transition
- scalar initializers
- array initializer

A Real-World Example


```
ss AICresponsibleStates {
  state st_process {
    int resp[MAXSEG][MAXERR] = {
      /* Segment * ERROR: blTs, blW, idTs, idW, d1ts, d1w, d2ts, d2w, landau, vacuum */
      /*****************************************************************************/
      /* seg[1]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[2]  */     {  2,    2,   2,   1,   1,    1,   1,    2,   1,      1},
      /* seg[3]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[4]  */     {  2,    2,   2,   1,   1,    1,   1,    1,   1,      1},
      /* seg[5]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[6]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
    };
    when(delay(1)) {
      int isE = 0, isM = 0;
      int s, e;
      for( s = 0; s < MAXSEG; s++ ) {
        if( seg[s] == 1 ) {        /* segments: 0=ok */
          for( e = 0; e < MAXERR; e++ ) {
            if( err[e] == 0 ) {    /* errors: 1=ok */
              if( resp[s][e] == 1 ) {
                isM = 1;
              }
              if( resp[s][e] == 2 ) {
                isE = 2;
              }
            }
          }
        }
      }
      AICresponsible = isE + isM;
      pvPut(AICresponsible);
    } state st_process
  }
}
```

- declaration local to state
- (multi-variable) declarations local to transition
- scalar initializers
- array initializer

A Real-World Example


```
ss AICresponsibleStates {
  state st_process {
    int resp[MAXSEG][MAXERR] = {
      /* Segment * ERROR: blTs, blW, idTs, idW, d1ts, d1w, d2ts, d2w, landau, vacuum */
      /*****************************************************************************/
      /* seg[1]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[2]  */     {  2,    2,   2,   1,   1,    1,   1,    2,   1,      1},
      /* seg[3]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[4]  */     {  2,    2,   2,   1,   1,    1,   1,    1,   1,      1},
      /* seg[5]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
      /* seg[6]  */     {  2,    2,   2,   2,   1,    1,   1,    1,   1,      1},
    };
    when(delay(1)) {
      int isE = 0, isM = 0;
      int s, e;
      for( s = 0; s < MAXSEG; s++ ) {
        if( seg[s] == 1 ) {        /* segments: 0=ok */
          for( e = 0; e < MAXERR; e++ ) {
            if( err[e] == 0 ) {    /* errors: 1=ok */
              if( resp[s][e] == 1 ) {
                isM = 1;
              }
              if( resp[s][e] == 2 ) {
                isE = 2;
              }
            }
          }
        }
      }
      AICresponsible = isE + isM;
      pvPut(AICresponsible);
    } state st_process
  }
}
```

- declaration local to state
- (multi-variable) declarations local to transition
- scalar initializers
- array initializer

Outline

5 Appendix
- Types and Declarations
- Declaration Example
- Rationale for Safe Mode
- Procedures

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Safe Mode: Rationale

- Why not simply lock variables?
- No programmer intervention should be needed
- Simultaneously lock all variables used in an action block during the execution of that block
- What if action code calls `epicsThreadSleep`?
- Locking would prevent
  - other state sets that use these variables from proceeding
  - the CA callback thread from proceeding
  - Note: there is only one CA callback thread for all SNL programs on the same IOC

return to main talk

Outline

5 Appendix
- Types and Declarations
- Declaration Example
- Rationale for Safe Mode
- Procedures

Procedures: Parameterizable State Sets

- Rationale
  - add abstraction to SNL
  - avoid repetition, reduce cut and paste
  - obviate the need for escape to C code
- Design
  - syntactically like a state set with parameters
  - but runtime behavior is rather like a procedure call, i.e. synchronous
  - calling a procedure
    - `call procedure_name(arg1, arg2, ...);`
  - returning from a procedure
    - `when(...) {...} return`
- Implementation
  - not easy, needs deep refactoring
  - don't hold your breath

Procedures: Parameterizable State Sets

- Rationale
  - add abstraction to SNL
  - avoid repetition, reduce cut and paste
  - obviate the need for escape to C code
- Design
  - syntactically like a state set with parameters
  - but runtime behavior is rather like a procedure call, i.e. synchronous
  - calling a procedure
    - `call procedure_name(arg1, arg2, ...);`
  - returning from a procedure
    - `when(...) {...} return`
- Implementation
  - not easy, needs deep refactoring
  - don't hold your breath

Procedures: Parameterizable State Sets

- Rationale
  - add abstraction to SNL
  - avoid repetition, reduce cut and paste
  - obviate the need for escape to C code
- Design
  - syntactically like a state set with parameters
  - but runtime behavior is rather like a procedure call, i.e. synchronous
  - calling a procedure
    - `call procedure_name(arg1, arg2, ...);`
  - returning from a procedure
    - `when(...) {...} return`
- Implementation
  - not easy, needs deep refactoring
  - don't hold your breath

Procedures: Parameterizable State Sets

- Rationale
  - add abstraction to SNL
  - avoid repetition, reduce cut and paste
  - obviate the need for escape to C code
- Design
  - syntactically like a state set with parameters
  - but runtime behavior is rather like a procedure call, i.e. synchronous
  - calling a procedure
    - `call procedure_name(arg1, arg2, ...);`
  - returning from a procedure
    - `when(...) {...} return`
- Implementation
  - not easy, needs deep refactoring
  - don't hold your breath

Procedures: Parameterizable State Sets

- Rationale
  - add abstraction to SNL
  - avoid repetition, reduce cut and paste
  - obviate the need for escape to C code
- Design
  - syntactically like a state set with parameters
  - but runtime behavior is rather like a procedure call, i.e. synchronous
  - calling a procedure

```
call procedure_name(arg1, arg2, ...);
```

  - returning from a procedure
    - `when(...) {...} return`
- Implementation
  - not easy, needs deep refactoring
  - don't hold your breath

Procedures: Parameterizable State Sets

- Rationale
  - add abstraction to SNL
  - avoid repetition, reduce cut and paste
  - obviate the need for escape to C code
- Design
  - syntactically like a state set with parameters
  - but runtime behavior is rather like a procedure call, i.e. synchronous
  - calling a procedure

```
call procedure_name(arg1, arg2, ...);
```

  - returning from a procedure

```
when(...)  {...} return
```

- Implementation
  - not easy, needs deep refactoring
  - don't hold your breath

Procedures: Parameterizable State Sets

- Rationale
  - add abstraction to SNL
  - avoid repetition, reduce cut and paste
  - obviate the need for escape to C code
- Design
  - syntactically like a state set with parameters
  - but runtime behavior is rather like a procedure call, i.e. synchronous
  - calling a procedure

```
call procedure_name(arg1, arg2, ...);
```

  - returning from a procedure

```
when(...)  {...} return
```

- Implementation
  - not easy, needs deep refactoring
  - don't hold your breath

return to main talk