State Notation Language and Sequencer Users' Guide

Release 2.2.6

William Lupton (wlupton@keck.hawaii.edu)
Benjamin Franksen (benjamin.franksen@helmholtz-berlin.de)

January 02, 2018

1 Introduction **1**
1.1 About . . . 1
1.2 Overview . . . 1
1.3 Acknowledgements . . . 2
1.4 Copying and Distribution . . . 2
1.5 Note on Versions . . . 3

2 Download and Installation **5**
2.1 Prerequisites . . . 5
2.2 Download . . . 5
2.3 Unpack . . . 6
2.4 Configure and Build . . . 6
2.5 Test . . . 7
2.6 Use . . . 8
2.7 Report Bugs . . . 8
2.8 Contribute . . . 8

3 Tutorial **11**
3.1 A First Example . . . 11
3.2 Variables . . . 12
3.3 Built-in PV Functions . . . 13
3.4 A Complete Program . . . 13
3.5 Adding a Second State Set . . . 14
3.6 Variable Initialization and Entry Blocks . . . 15
3.7 PV Names Using Program Parameters . . . 16
3.8 Data Types . . . 16
3.9 Arrays of Variables . . . 17
3.10 Dynamic Assignment . . . 17
3.11 Status of Process Variables . . . 18
3.12 Synchronizing State Sets with Event Flags . . . 18
3.13 Queuing Monitors . . . 19
3.14 Asynchronous Use of pvGet . . . 20
3.15 Asynchronous Use of pvPut . . . 20
3.16 Connection Management . . . 21
3.17 Multiple Instances and Reentrant Object Code . . . 21
3.18 Process Variable Element Count . . . 22
3.19 What's Happening at Run Time . . . 22
3.20 Safe Mode . . . 22
3.21 Common Pitfalls and Misconceptions . . . 24

4 Compiling SNL Programs **25**
4.1 The SNL to C Compiler . . . 25
4.2 C Pre-processor . . . 27
4.3 Complete Build . . . 27
4.4 Building a Stand-alone Program . . . 28

4.5 Using makeBaseApp . . . 29

5 Running SNL Programs **31**
5.1 Command Syntax . . . 31
5.2 Starting a Program . . . 31
5.3 Program Parameters . . . 32
5.4 Examining a Program . . . 32
5.5 Shell Command Reference . . . 33

6 Escape to C Code **37**
6.1 Preprocessor Directives . . . 37
6.2 Defining C Functions within the Program . . . 38
6.3 Calling PV Functions from C . . . 38
6.4 Variable Modification for Reentrant Option . . . 38

7 SNL Reference for Version 2.2 **41**
7.1 Lexical Syntax . . . 41
7.2 Program . . . 42
7.3 Definitions . . . 44
7.4 State Set . . . 52
7.5 Statements and Expressions . . . 55
7.6 Built-in Constants . . . 58
7.7 Built-in Functions . . . 60
7.8 Safe Mode . . . 69
7.9 Foreign Variables and Functions . . . 70

8 Examples of SNL Programs **71**
8.1 Entry and exit action example . . . 71
8.2 Dynamic assignment example . . . 71
8.3 Complex example . . . 72

9 Release Notes for Version 2.2 **77**
9.1 Release 2.2.5 . . . 77
9.2 Release 2.2.4 . . . 77
9.3 Release 2.2.3 . . . 78
9.4 Release 2.2.2 . . . 79
9.5 Release 2.2.1 . . . 80

10 Known Problems **89**
10.1 Known Problems in Release 2.2.5 . . . 89
10.2 Known Problems in Release 2.2.4 . . . 89
10.3 Known Problems in Release 2.2.3 . . . 89
10.4 Known Problems in Release 2.2.2 . . . 89
10.5 Known Problems in Release 2.2.1 . . . 89

11 Plans for the Future **91**

12 Glossary of Terms **93**

Index **95**

CHAPTER

ONE

INTRODUCTION

1.1 About

This project is a component of the *Experimental Physics and Industrial Control System* (EPICS), a set of tools, libraries and applications developed collaboratively and used worldwide to create distributed soft real-time control systems for large scale scientific instruments.

The *State Notation Language* (SNL), is a domain specific programming language that smoothly integrates with and depends and builds on EPICS base. This project defines SNL and provides an implementation, consisting of the SNL compiler and runtime system.

The Sequencer is free software, licensed under the EPICS Open License.

For further information

- take a look at the *Overview* to get a first impression of the Sequencer's features.
- This talk about version 2.1 of the Sequencer contains a short introduction to SNL, including a walk through a simple example program to illustrate the basic concepts.
- A (slightly out-dated) *Tutorial* is also available,
- as well as a detailed *SNL Reference for Version 2.2*,
- and instructions for download and *Download and Installation*.
- All of these are part of the *Users' Guide*.

This page describes version 2.2 and the documentation, in particular the language reference, has been updated to reflect the *changes between 2.1 and 2.2*. For older versions, please refer to the old version of this page.

1.2 Overview

In SNL you structure your program as a set of concurrently running finite state machines, called `state_sets`. State sets are declared by listing their `states`, which in turn define under which condition (`when`) the state set reaches another state and what the program should do once the transition is triggered.

SNL allows you to bind program variables to externally defined *process variables* (PVs), such as provided by an EPICS runtime database. This can be done in such a way that the value of the program variable gets continuously updated whenever the value of the acssociated PV changes. Such variables can then be used inside the state transition conditions, and the runtime system takes care that the conditions are evaluated when and only when changes to the associated PV occur.

In the code that gets executed when a state transition takes place, you can explicitly read the value of the PV into the associated program variable (`pvGet`), or to send the value of the variable to update the PV (`pvPut`). (These are just the two most important of many more *Built-in Functions*.)

The mechanism behind all this PV magic is the EPICS Channel Access network protocol with its support for subscriptions. However, when programming in SNL you need not concern yourself with the details of the mechanism. The SNL compiler and runtime system manage all this behind the scenes, and also take care that the integrity of

your variables is maintained, even in the presence of multiple concurrent state sets inside a single program (at least in *Safe Mode*). *Programming in SNL is free of locks and therefore also free of deadlocks.*

Despite all this, SNL is emphatically *not* a high-level language: most of its syntax and semantics are directly inherited from C. The language is designed for seamless integration with C code; for instance, you can directly call C procedures in SNL action blocks and state transition conditions, and the types of variables you can declare in SNL map directly to C types. The compiler for SNL (called `snc`, see *Compiling SNL Programs*) takes a very minimalistic approach to compilation by generating portable standard C89/90 code that works on all platforms supported by EPICS. In fact, most of the SNL action code is compiled almost verbatim to C, and the generated code is quite human readable. The SNL compiler also delegates all type checking to the C compiler (and some care has been taken to generate the code in such a way that the C type checker has actually enough information to *do* useful type checking).

Compilation of the generated code and linking to the SNL runtime system is also *not* done by snc. Instead this task is delegated to the EPICS build system by making the Sequencer an EPICS *support module* that automatically adds appropriate build rules for SNL source files.

1.3 Acknowledgements

The sequencer was originally developed by Andy Kozubal at Los Alamos National Laboratory (LANL). It was subsequently modified by William Lupton, formerly at the W. M. Keck Observatory (Keck), with contributions by Greg White of the Stanford Linear Accelerator Center National Accelerator Laboratory (SLAC-NAL).

Eric Norum, Janet Anderson, and Marty Kraimer (APS) made the initial port to EPICS 3.14 that led to the 2.0.x series. Eric Norum and Andrew Johnson (APS) provided lots of patches up to release 2.0.12.

Mark Rivers made a number of useful suggestions and tested version 2.1 with several sets of existing programs on various platforms. J. Lewis Muir provided documentation patches.

1.4 Copying and Distribution

4. You may modify a copy or copies of the Software or any portion of it, thus forming a work based on the Software, and distribute copies of such work outside your organization, if you meet all of the following conditions:

   (a) Copies in source code must include the copyright notice and this Software License Agreement;

   (b) Copies in binary form must include the copyright notice and this Software License Agreement in the documentation and/or other materials provided with the copy;

   (c) Modified copies and works based on the Software must carry prominent notices stating that you changed specified portions of the Software.

5. Portions of the Software resulted from work developed under a U.S. Government contract and are subject to the following license: the Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license in this computer software to reproduce, prepare derivative works, and perform publicly and display publicly.

6. WARRANTY DISCLAIMER. THE SOFTWARE IS SUPPLIED "AS IS" WITHOUT WARRANTY OF ANY KIND. THE COPYRIGHT HOLDERS, THEIR THIRD PARTY LICENSORS, THE UNITED STATES, THE UNITED STATES DEPARTMENT OF ENERGY, AND THEIR EMPLOYEES: (1) DISCLAIM ANY WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON-INFRINGEMENT, (2) DO NOT ASSUME ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF THE SOFTWARE, (3) DO NOT REPRESENT THAT USE OF THE SOFTWARE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS, (4) DO NOT WARRANT THAT THE SOFTWARE WILL FUNCTION UNINTERRUPTED, THAT IT IS ERROR-FREE OR THAT ANY ERRORS WILL BE CORRECTED.

7. LIMITATION OF LIABILITY. IN NO EVENT WILL THE COPYRIGHT HOLDERS, THEIR THIRD PARTY LICENSORS, THE UNITED STATES, THE UNITED STATES DEPARTMENT OF ENERGY, OR THEIR EMPLOYEES: BE LIABLE FOR ANY INDIRECT, INCIDENTAL, CONSEQUENTIAL, SPECIAL OR PUNITIVE DAMAGES OF ANY KIND OR NATURE, INCLUDING BUT NOT LIMITED TO LOSS OF PROFITS OR LOSS OF DATA, FOR ANY REASON WHATSOEVER, WHETHER SUCH LIABILITY IS ASSERTED ON THE BASIS OF CONTRACT, TORT (INCLUDING NEGLIGENCE OR STRICT LIABILITY), OR OTHERWISE, EVEN IF ANY OF SAID PARTIES HAS BEEN WARNED OF THE POSSIBILITY OF SUCH LOSS OR DAMAGES.

1.5 Note on Versions

This manual describes version 2.2 of the sequencer. This version adds support for declaring variables of almost all types expressible in C. It also adds support for defining functions and struct types in SNL. The backend of the compiler was changed so that generated names follow a consistent naming convention. The interface between compiler and runtime system has been separated from the public API available to escaped C code.

Version 2.1 added support for local definitions (including variable declarations) at all levels, a significantly improved compiler that employs new lexer and parser generators, lots of bug fixes, and a new *safe mode* to avoid variable corruption due to race conditions.

Version 2.0 differs from version 1.9 mainly in that sequencer run-time code can run under any operating system for which an EPICS OSI (Operating System Independent) layer is available, and message systems other than channel access can be used. It depends on libraries which are available only with EPICS R3.14.

An interim version 1.9.4 was made available to the EPICS community; all new developments apart from major bug fixes will be based on version 2.0.

Version 1.9 was written by Andy Kozubal, the original author of this software. This version of the manual describes version 2.0, for which the changes have been implemented by William Lupton of W. M. Keck Observatory and Greg White of Stanford Linear Accelerator Center (SLAC).

1.5.1 Versioning Policy

Starting with release 2.0.0, the third digit is the patch level and will be incremented each time a new version is released, no matter how minor the changes (except if it is zero). The second digit is the minor release number and will be incremented each time functional changes are made. The first digit is the major release number and will be incremented only when major changes are made.

Starting with version 2.1, the following refined rules apply:

- Changes in the major (first) number indicate disruptive changes that may break backward compatibility. When upgrading to a new major version, existing programs may need to be reviewed and possibly adapted to changes in syntax or semantics.
- An increase of the minor (second) release number indicates the availability of new features, while source code compatibility for existing programs is preserved. However, such changes may involve major reorganizations of the implementation, und therefore introduce new bugs. Care should be taken when upgrading to a new minor release, especially if the new patch-level number is zero: these should be used in production systems only after testing that everything still works as expected.

  Major and minor release number together are referred to as the sequencer's *version*.
- Changes in the patch-level (third) release number are strictly reserved for bug-fixes, including fixes and extensions to the documentation and the tests. No new features will appear, except where unavoidable to cleanly fix a bug. It should always be safe (and is recommended) to upgrade to the latest patch level release.
- A patch-level release number of zero indicates a pre-release. Such releases are still in the process of stabilizing and thus might receive significant changes if it turns out that this would be better in the long run. A fourth number may indicate progress toward the first stable release, indicated by a patch-level release number of one.

CHAPTER

TWO

DOWNLOAD AND INSTALLATION

This chapter gives instructions on how to download, build, and install the sequencer, as well as how to run the tests, build the manual, report bugs, and contribute patches.

2.1 Prerequisites

You need to have EPICS base and its dependencies (GNU make, Perl) installed. Any version of EPICS base starting with 3.14.12.2 and up to 3.15 should do.

From version 2.1 onwards, building the sequencer requires the lexer generator tool re2c. If you are on a Linux system, you will probably want to use the re2c package your distribution provides, otherwise sources and windows binaries are available from the re2c download page. The minimum version required is 0.9.9, but I recommend using the latest version that is avaliable for your system.

2.2 Download

Releases are available here:

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

The current stable release is 2.2.5. Please take a look at the *Versioning Policy* if you are unsure whether to upgrade to a new release.

Note: The documentation and download links you will find on this page are for version 2.2. The documentation for version 2.1 is still there and will be maintained at least until version 2.3 is released.

Development snapshots are available under the name

> seq-<version>-snapshot-<date>.tar.gz

where <version> is the branch name without the 'branch-' prefix.

In the releases directory there is always a symbolic link to the latest snapshot.

Here are all releases for version 2.2:

Version	Release Notes	Known Problems
2.2.5	*Release 2.2.5*	n/a
2.2.4	*Release 2.2.4*	*Known Problems in Release 2.2.4*
2.2.3	*Release 2.2.3*	*Known Problems in Release 2.2.3*
2.2.2	*Release 2.2.2*	*Known Problems in Release 2.2.2*
2.2.1	*Release 2.2.1*	*Known Problems in Release 2.2.1*

If you want to help testing, please use the latest snapshot, or check out the darcs repository for branch-2-2:

```
darcs get http://www-csr.bessy.de/control/SoftDist/sequencer/repo/branch-2-2
```

Since release 2.2.2 there is also a git mirror, so you can as well

```
git clone http://www-csr.bessy.de/control/SoftDist/sequencer/repo/branch-2-2.git
```

but please note that this is just a mirror, no more: I cannot accept git patches (or whatever they are called in git); it may also happen that I have to re-create the whole git repo from scratch (because incremental conversion sometimes fails).

Development of new features now happens on branch-2-3.

See Contribute for a short description how to record and send patches.

You can also follow development by using the repository browser.

2.3 Unpack

Change to the directory that you wish to be the parent of the sequencer source tree. Then unpack and untar the file. For example:

```
gunzip seq-x.y.z.tar.gz
tar xf seq-x.y.z.tar
```

or, if you have GNU tar, simply:

```
tar zxf seq-x.y.z.tar.gz
```

You can now:

```
cd seq-x.y.z
```

and look at the source tree. The actual source code for the sequencer is under `src`. The documentation sources are under `documentation` and consist of plain readable text files (actually, the format is reStructuredText but you need to know about that only if you plan to make changes to the docs).

In what follows, `$SEQ` refers to the directory where you are now, i.e. `.../seq-x.y.z/`.

2.4 Configure and Build

The sequencer uses the EPICS build system. This means there is no automatic configuration, instead you have to edit the file `configure/RELEASE` and perhaps also `configure/CONFIG_SITE`. These are make include files, so the syntax is that of (GNU) make.

In `configure/RELEASE`, change the definition of the variable `EPICS_BASE` to the path where your EPICS base is installed.

In `configure/CONFIG_SITE`, you can specify the target architectures for which to build via the `CROSS_COMPILER_TARGET_ARCHS` variable (a subset of those for which EPICS has been built, default is all). You can also configure where the re2c tool is installed (the default configuration assumes that it can be found in your `PATH`).

Your environment should be configured for building EPICS applications. This means that `EPICS_HOST_ARCH` and (possibly) `LD_LIBRARY_PATH` should be correctly defined. See the EPICS Application Developer's Guide for details.

After changing the files in `configure`, run GNU make.

Note that make builds first in the `configure` directory, then the `src` tree, and finally the `test` and `examples` trees. A failure in the latter two will not impact your ability to write SNL programs (but is still a bug and should be reported, see Report Bugs).

2.4.1 Building the Manual

From 2.0.99 on, the manual is in reStructuredText format. This format is (more or less) readable plain text, so this section is optional.

Building the manual means generating a set of html pages and maybe a single pdf from the sources.

The html pages are generated by issuing:

```
make html
```

This will generate the home page and install it into the directory `$SEQ/html`. This step requires that you have Python and Sphinx installed on your system.

If, in addition, you want a printable version (pdf), do:

```
make docs
```

This generates a pdf file named `Manual.pdf` and also puts it into the `html` subdirectory. Note that pdf generation is done via latex, so you need to have a working latex installation. On my system (kubuntu karmic at the time of writing this) I also needed to install the package tetex-extra.

2.5 Test

You can run the automated test suite by issuing

```
make -s runtests
```

(the -s option is to suppress irrelevant output generated by make).

To run just one test from the suite, switch to the build directory of the test that corresonds to your `EPICS_HOST_ARCH` and execute perl with the name of the test program ending in `.t`. For example, to run the evflag test on a linux-x86_64 host, change directory to `test/validate/O.linux-x86_64` and run

```
perl evflag.t
```

There are two ways to run a test that has an associated database (evflag is such a test). The above will run the state machine and database on the same IOC. To run the test as two separate IOCs, use the test program ending in `Ioc.t` (e.g. evflagIoc.t). The IOC running the state machine runs in the foreground, and the one running the database runs in the background.

The test suite can also be run on an embedded system. Currently only vxWorks systems are supported. To do this, point the vxWorks startup script to

```
$SEQ/test/validate/O.$T_A/st.cmd
```

where `$T_A` is the name of the target architecture and `$SEQ` refers to the (absolute) path of your sequencer installation. The system will start an IOC and will run a number of SNL test programs, one after the other, after each one giving a summary of how many tests failed etc.

To check out an example, change directory to examples/demo and run

```
../../bin/$EPICS_HOST_ARCH/demo stcmd.host
```

The output should look something like this:

```
ben@sarun[1]: .../examples/demo > ../../bin/linux-x86/demo stcmd.host
dbLoadDatabase "../../dbd/demo.dbd"
demo_registerRecordDeviceDriver(pdbbase)
dbLoadRecords "demo.db"
iocInit
Starting iocInit
############################################################################
###  EPICS IOC CORE built on Mar  3 2010
```

```
###  EPICS R3.14.8.2 $R3-14-8-2$ $2006/01/06 15:55:13$
################################################################################
iocInit: All initialization complete
seq &demo "debug=0"
SEQ Version 2.1.0, compiled Fri Jul 15 12:44:09 2011
Spawning sequencer program "demo", thread 0x8064f48: "demo"
start -> ramp_up
epics> light_off -> light_on
ramp_up -> ramp_down
light_on -> light_off
ramp_down -> ramp_up
light_off -> light_on
ramp_up -> ramp_down
...
```

If you see the "start -> ramp_up" etc. messages, things are good.

2.6 Use

This is a short description how to use the sequencer in an EPICS application. For more general usage information, see the section on *Compiling SNL Programs* and *Running SNL Programs*.

To use the sequencer in an EPICS application, change the definition of `SNCSEQ` in `configure/RELEASE` (that is, the one in your application, not the sequencer's) to contain the path to your sequencer installation.

As soon as `SNCSEQ` is defined, the EPICS build system automagically includes the build rules defined in the sequencer. To add an SNL program to your application, write something like

```
SRCS += xyz.st
abc_LIBS += seq pv
```

into your Makefile. Here, `xyz.st` is the name of your SNL program, and `abc` is the name of the library or binary to produce. Note that `.st` files are run through the C preprocessor (`cpp`) before giving them to the SNL compiler. Use the extension `.stt` to avoid this. For details, see Chapter 4 of the EPICS Application Developer's Guide.

A complete example application that also uses the sequencer can be produced using makeBaseApp, e.g.

```
makeBaseApp.pl -t example ex
```

Take a look at `exApp/src`, especially the `Makefile`.

2.7 Report Bugs

Please send bug reports to to tech-talk@aps.anl.gov or the maintainer (currently this is me). It helps if you include which release of the sequencer and EPICS base release you are using.

2.8 Contribute

I am always happy to receive patches (bug fixes, improvements, whatever). For minor changes you don't need to bother with darcs, just send me a patch file.

For more involved changes you might want to send a darcs patch. You can create a local copy of the darcs repository (the stable branch in this example) by saying:

```
darcs get http://www-csr.bessy.de/control/SoftDist/sequencer/repo/stable
```

Assuming you have made some changes, first update your repository to include the latest changes from upstream (with darcs this is not strictly necessary, but good practice as it helps to avoid unnecessary conflicts):

```
darcs pull
```

(darcs will ask you for each patch that is not yet in your repo). Then record your changes (if you haven't already):

```
darcs record
```

(darcs will prompt you for every single change you made and then prompt you for giving the patch a name). Finally say:

```
darcs send
```

A word of caution: it may happen that I darcs obliterate patches in the experimental branch. If darcs send asks whether to include patches that you don't have authored, this is probably what happened. In that case can (but you need not necessarily) quit and obliterate them in your source tree, too, before trying to darcs send again.

Please respect the coding style when making changes. This includes indentation (tabs or spaces, how many) and all the other little things on which programmers like to differ ;-) like placement of braces etc. Note that for historical reasons the style differs somewhat between files and subdirectories. It is much easier for me to review patches if they do not contain gratuitous changes or combine several unrelated changes in a single patch.

Also, please take care that your patch does not accidentally contain site-specific changes (typically done in configure). For my own work, I usually record such changes with a description that contains 'DONT SEND THIS' or something similar, so I don't accidentally record them together with other changes.

CHAPTER

THREE

TUTORIAL

This chapter gives a gentle introduction to State Notation Language (SNL).

3.1 A First Example

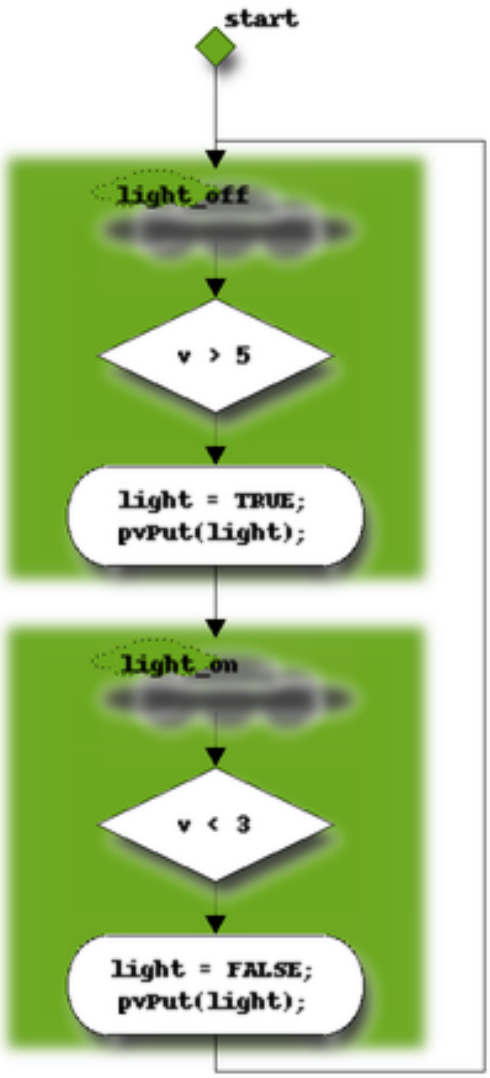
We start with a simple state machine `volt_check` that controls a light switch depending on the value of a voltage measurement and the internal state of the program. The following code fragment defines the state machine:

```
ss volt_check {
  state light_off {
    when (voltage > 5.0) {
      light = TRUE;
      pvPut(light);
    } state light_on
  }
  state light_on {
    when (voltage < 3.0) {
      light = FALSE;
      pvPut(light);
    } state light_off
  }
}
```

At the top level we use the keyword `ss` to declare a *state set* (which is SNL speak for state machine) named `volt_check`. Inside the code block that follows, we define the *states* of this state set, using the `state` keyword. There are two states here: `light_off` and `light_on`. Inside each state, we define conditions under which the program will enter another state, indicated by the keyword `when`. The block following the condition contains *action statements* that are executed when the condition fires.

In our example, in state `light_off`, whenever the voltage exceeds a value of 5.0, the light switch is turned on, and the internal state changes to `light_on`. In state `light_on`, whenever the voltage is or drops below 3.0, the light switch is turned off, and the internal state changes to `light_off`.

The following is a graphical representation of the above state machine:



Note that the output or action depends not only on the input or condition, but also on the current state. For instance, an input `voltage` of 4.2 volts does not alone determine the output (`light`), the current state matters, too.

As you can see, the SNL code is syntactically very similar to the C language. Particularly, the syntax for variable declarations, expressions, and statements are exactly as in C, albeit with a few restrictions.

You might wonder about the function calls in the above code. The `pvPut` function is a special built-in function that writes (or puts) the value in the variable `light` to the appropriate process variable. But before I can explain how this works, we must talk about how program variables are "connected" to process variables.

3.2 Variables

SNL programs interact with the outside world via variables that are bound to (or connected to) `process variables` (PVs) in EPICS. In our example, there are two such variables: `voltage`, which represents a measured voltage, and `light` which controls a light switch. In an actual SNL program, these variables must be declared before they can be used:

```
float voltage;
int light;
```

We also want to associated them with PVs i.e. EPICS record fields:

```
assign voltage to "Input_voltage";
assign light to "Indicator_light";
```

The above `assign` clauses associate the variables `voltage` and `light` with the process variables "Input_voltage" and "Indicator_light" respectively.

We also want the value of `voltage` to be updated automatically whenever it changes. This is accomplished with the following code:

```
monitor voltage;
```

Whenever the value in the control system (the EPICS database) changes, the value of `voltage` will likewise change. Note however that this depends on the underlying system sending update messages for the value in question. When and how often such updates are communicated by the underlying system may depend on the configuration of the PV. For instance if the PV "Input_voltage" is the VAL field of an ai (analog input) record, then the value of the MDEL field of the same record specifies the amount of change that the designer considers a "relevant" change; smaller changes will not cause an event to be sent, and accordingly will not cause a state change in the above program.

3.3 Built-in PV Functions

I said above that the program interacts with the outside world via variables assigned to PVs. However, mutating such a variable e.g. via the C assignment operator (see *Assignment Operators*), as in:

```
light = TRUE;
```

only changes the value of `light` as seen from inside the program. In order for the new value to take effect, it must be written to the PV connected with the variable. This is done by calling the special built-in function `pvPut`, that gets the variable as argument.

Note that calling such a special built-in function does *not* follow the standard C semantics for function calls! Particularly, what actually gets passed to the function is not the *value* of the variable `light` (as it would in C), instead an internal representation of the variable gets passed (by reference). You can think of what actually gets passed as an "object" (as in "object-oriented") or a "handle" that contains all the necessary run-time information, one of which is the name of PV the variable is connected with.

There are many more of these built-in functions, the *SNL Reference for Version 2.2* contains detailed description of each one. For now, let's keep to the basics; I'll mention just one more built-in function: With `pvGet`, you can poll PVs explicitly, instead of using `monitor`. That is, a statement such as

```
pvGet(voltage);
```

has the effect of sending a get request to the PV "Input_voltage", waiting for the response, and then updating the variable with the new value.

3.4 A Complete Program

Here is what the complete program for our example looks like:

```
program level_check

float voltage;
assign voltage to "Input_voltage";
monitor voltage;

short light;
assign light to "Indicator_light";
```

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

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

Each program must start with the word `program`, followed by the name of the program (an identifier):

```
program level_check
```

After that come global declarations and then one or more state sets.

3.5 Adding a Second State Set

We will now add a second state set to the previous example. This new state set generates a changing value as its output (a triangle function with amplitude 11).

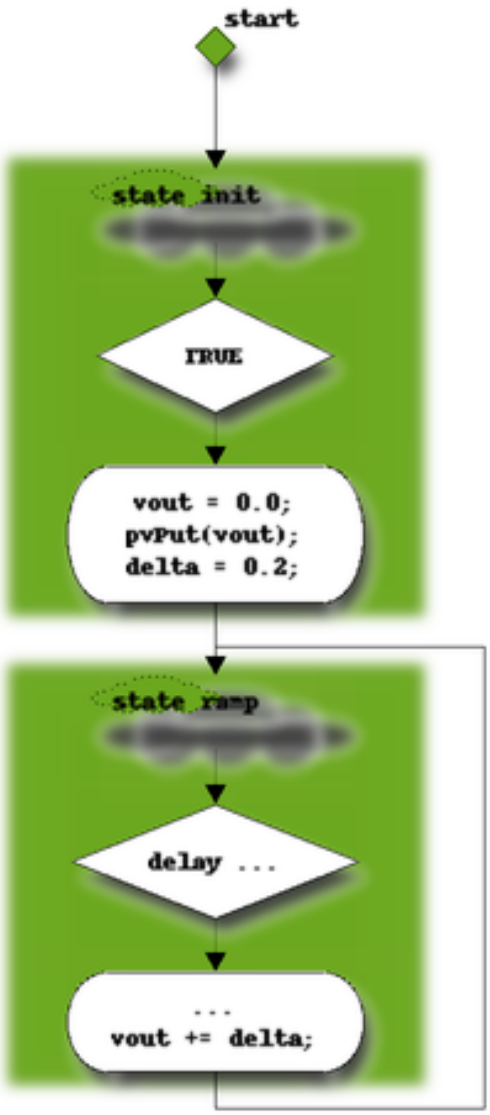
First, we add the following lines to the declaration:

```
float vout;
float delta;
assign vout to "Output_voltage";
```

Next we add the following lines after the first state set:

```
ss generate_voltage {
  state init {
    when () {
      vout = 0.0;
      pvPut(vout);
      delta = 0.2;
    } state ramp
  }
  state ramp {
    when (delay(0.1)) {
      if ((delta > 0.0 && vout >= 11.0) \|\|
        (delta < 0.0 && vout <= -11.0)) {
        delta = -delta; /* change direction */
      }
      vout += delta;
    } state ramp;
  }
}
```

The above example exhibits several concepts. First, note that the `transition` clause in state `init` contains an empty event expression. This means unconditional execution of the transition. The first state in each state set is always the initial state, so we give it the name `init`. From this first state there is an immediate unconditional transition to the state `ramp`, initializing some variables during the transition. Note that the `ramp` state always returns to itself. The structure of this state set is shown in the following STD:



The final concept introduced in the last example is the `delay` function. This function returns a boolean that tells us whether the given time interval has elapsed. The interval is given in seconds (as a floating point value) and counts from the time the state was entered.

At this point, you may wish to try an example with the two state sets. You can jump ahead and read parts of the Chapters *Compiling SNL Programs* and *Running SNL Programs* to find out how. You probably want to pick unique names for your process variables, rather than the ones used above.

3.6 Variable Initialization and Entry Blocks

Since version 2.1 it has become simpler to initialize variables: you can use the same syntax as in C, i.e. initialize together with the declaration:

```
float vout = 0.0;
float delta = 0.2;
```

which, by the way, can also be written as

```
float vout = 0.0, delta = 0.2;
```

More complicated initialization (e.g. involving non-constant expressions or side-effects) can be done using an `entry` block instead of using a separate state:

```
ss generate_voltage {
  state ramp {
    entry {
      pvPut(vout);
    }
    when (delay(0.1)) {
      ...
    } state ramp;
  }
}
```

The actions in an entry block in a state declaration are executed whenever the state is entered from a different state. In this case this means the

```
pvPut(vout);
```

that appears inside the entry block will be executed only once when the state is entered for the first time.

3.7 PV Names Using Program Parameters

You can use program parameter substitution to parameterize the PV names in your program. In our example we could replace the `assign` statements with the following:

```
assign voltage to "{unit}:ai1";
assign vout to "{unit}:ao1";
```

The string within the curly braces is the name of a program parameter and the whole thing (the name and the braces) are replaced with the value of the parameter. For example, if the parameter "unit" has value "DTL_6:CM_2", then the expanded PV name is "DTL_6:CM_2:ai1". See *Program Parameters* for more on program parameters (and particularly how to give them values).

3.8 Data Types

In earlier versions, variables were restricted to a hand full of predefined types, plus one or two-dimensional arrays of these.

This is no longer true: you can declare variables of any type you like. The only restrictions are:

1. you cannot *define* new types, only use them in declarations
2. when using type aliases ("typedef") you must prefix them with the keyword "typename"
3. only variables of the above mentioned restricted list can be `assign`'ed to PVs.

The built-in types are: `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float` , and `double`. These correspond exactly to their C equivalents. In addition there is the type `string`, which is an array of 40 `char`.

Sequencer variables having any of these types may be assigned to a process variable. The type declared does not have to be the same as the native control system value type. The conversion between types is performed at run-time. For more details see the *corresponding section in the reference*.

You may specify array variables as follows:

```
long arc_wf[1000];
```

When assigned to a process variable, operations such as `pvPut` are performed for the entire array.

3.9 Arrays of Variables

Often it is necessary to have several associated process variables. The ability to assign each element of an SNL array to a separate process variable can significantly reduce the code complexity. The following illustrates this point:

```
float Vin[4];
assign Vin[0] to "{unit}1";
assign Vin[1] to "{unit}2";
assign Vin[2] to "{unit}3";
assign Vin[3] to "{unit}4";
```

We can then take advantage of the `Vin` array to reduce code size as in the following example:

```
for (i = 0; i < 4; i++) {
  Vin[i] = 0.0;
  pvPut (Vin[i]);
}
```

We also have a shorthand method for assigning channels to array elements:

```
assign Vin to { "{unit}1", "{unit}2", "{unit}3", "{unit}4" };
```

Similarly, the monitor declaration may be either by individual element:

```
monitor Vin[0];
monitor Vin[1];
monitor Vin[2];
monitor Vin[3];
```

Alternatively, we can do this for the entire array:

```
monitor Vin;
```

And the same goes when Synchronizing State Sets with Event Flags and *Queuing Monitors*.

Double subscripts offer additional options:

```
double X[2][100];
assign X to {"apple", "orange"};
```

The declaration creates an array with 200 elements. The first 100 elements of `X` are assigned to (array) "apple", and the second 100 elements are assigned to (array) "orange" .

It is important to understand the distinction between the first and second array indices here. The first index defines a 2-element array of which each element is associated with a process variable. The second index defines a 100-element double array to hold the value of each of the two process variables. When used in a context where a number is expected, both indices must be specified, e.g. `X[1][49]` is the 50th element of the value of "orange" . When used in a context where a process variable is expected, e.g. with `pvPut`, then only the first index should be specified, e.g. `X[1]` for "orange" .

3.10 Dynamic Assignment

You may dynamically assign or re-assign variable to process variables during the program execution as follows:

```
float Xmotor;
assign Xmotor to "Motor_A_2";
...
sprintf (pvName, "Motor_%s_%d", snum, mnum)
pvAssign (Xmotor[i], pvName);
```

Note that dynamic (re-)assignment fails (with a compiler error) if the variable has not been assigned statically.

An empty string in the assign declaration implies no initial assignment and can be used to mark variables or array elements for later dynamic assignment:

```
assign Xmotor to "";
```

Likewise, an empty string can de-assign a variable:

```
pvAssign(Xmotor, "");
```

The current assignment status of a variable is returned by the `pvAssigned` function as follows:

```
isAssigned = pvAssigned(Xmotor);
```

The number of assigned variables is returned by the `pvAssignCount` function as follows:

```
numAssigned = pvAssignCount();
```

The following inequality will always hold:

```
pvConnectCount() <= pvAssignCount() <= pvChannelCount()
```

Having assigned a variable, you should wait for it to connect before using it (although it is OK to monitor it). See Connection Management.

3.11 Status of Process Variables

Process variables have an associated status, severity and time stamp. You can obtain these with the `pvStatus`, `pvSeverity` and `pvTimeStamp` functions. For example:

```
when (pvStatus(x_motor) != pvStatOK) {
printf("X motor status=%d, severity=%d, timestamp=%d\\n",
pvStatus(x_motor), pvSeverity(x_motor),
pvTimeStamp(x_motor).secPastEpoch);
...
```

These routines are described in *Built-in Functions*. The values for status and severity are defined in the include file *pvAlarm.h*, and the time stamp is returned as a standard EPICS `TS_STAMP` structure, which is defined in *tsStamp.h* . Both these files are automatically included when compiling sequences (but the SNL compiler doesn't know about them, so you will get warnings when using constants like `pvStatOK` or tags like `secPastEpoch`).

3.12 Synchronizing State Sets with Event Flags

State sets within a program may be synchronized through the use of event flags. Typically, one state set will set an event flag, and another state set will test that event flag within a `transition` clause. The `sync` statement may also be used to associate an event flag with a process variable that is being monitored. In that case, whenever a monitor is delivered, the corresponding event flag is set. Note that this provides an alternative to testing the value of the monitored channel and is particularly valuable when the channel being tested is an array or when it can have multiple values and an action must occur for any change.

This example shows a state set that forces a low limit always to be less than or equal to a high limit. The first `transition` clause fires when the low limit changes and someone has attempted to set it above the high limit. The second `transition` clause fires when the opposite situation occurs.

```
double loLimit;
assign loLimit to "demo:loLimit";
monitor loLimit;
evflag loFlag;
```

```
sync loLimit loFlag;

double hiLimit;
assign hiLimit to "demo:hiLimit";
monitor hiLimit;
evflag hiFlag;
sync hiLimit hiFlag;

ss limit {
  state START {
    when ( efTestAndClear( loFlag ) && loLimit > hiLimit ) {
      hiLimit = loLimit;
      pvPut( hiLimit );
    } state START

    when ( efTestAndClear( hiFlag ) && hiLimit < loLimit ) {
      loLimit = hiLimit;
      pvPut( loLimit );
    } state START
  }
}
```

The event flag is actually associated with the SNL variable, not the underlying process variable. If the SNL variable is an array then the event flag is set whenever a monitor is posted on any of the process variables that are associated with an element of that array.

3.13 Queuing Monitors

Neither testing the value of a monitored channel in a `transition` clause nor associating the channel with an event flag and then testing the event flag can guarantee that the sequence is aware of all monitors posted on the channel. Often this doesn't matter, but sometimes it does. For example, a variable may transition to 1 and then back to 0 to indicate that a command is active and has completed. These transitions may occur in rapid succession. This problem can be avoided by using the `syncq` statement to associate a variable with a queue. The `pvGetQ` function retrieves and removes the head of queue.

This example illustrates a typical use of `pvGetQ` : setting a command variable to 1 and then changing state as an active flag transitions to 1 and then back to 0. Note the use of `pvFlushQ` to clear the queue before sending the command. Note also that, if `pvGetQ` hadn't been used then the active flag's transitions from 0 to 1 and back to 0 might both have occurred before the `transition` clause in the `sent` state fired:

```
long command; assign command to "commandVar";

long active; assign active to "activeVar"; monitor active;
syncq active 2;

ss queue {
  state start {
    entry {
      pvFlushQ( active );
      command = 1;
      pvPut( command );
    }
    when ( pvGetQ( active ) && active ) {
    } state high
  }
  state high {
    when ( pvGetQ( active ) && !active ) {
    } state done
  }
  state done {
```

```
    /* ... */
  }
}
```

The `active` SNL variable could have been an array in the above example. It could therefore have been associated with a set of related control system `active` flags. In this case, the queue would have had an entry added to it whenever a monitor was posted on any of the underlying control system `active` flags.

3.14 Asynchronous Use of pvGet

Normally the `pvGet` operation completes before the function returns, thus ensuring data integrity. However, it is possible to use these functions asynchronously by specifying the `+a` compiler flag (see *Compiler Options*). The operation might not be initiated until the action statements in the current transition have been completed and it could complete at any later time. To test for completion use the function `pvGetComplete`, which is described in *Built-in Functions*.

`pvGet` also accepts an optional `SYNC` or `ASYNC` argument, which overrides the `+a` compiler flag. For example:

```
pvGet( initActive[i], ASYNC );
```

3.15 Asynchronous Use of pvPut

Normally `pvPut` is a "fire and forget" operation without any provisions for testing if and when it completed successfully. However, this behaviour can be modified by passing an optional `SYNC` or `ASYNC` argument. With `SYNC`, the call blocks until the operation is complete, while with `ASYNC` the call returns immediately. In the latter case, `pvPutComplete` tells you whether the operation completed.

For example,

```
pvPut(init[i], SYNC);
```

will block until the put operation to the PV behind `init[i]` (and all the processing resulting from it) is complete, while

```
pvPut(init[i], ASYNC);
```

does not block and instead lets you test completion explicitly, e.g.

```
when(pvPutComplete(init[i])) {
  ...
}
```

Note that `pvPutComplete` can only be used with single PVs. Testing completion for multiple PVs in a multi-PV array can be done with `pvArrayPutComplete` as in the following example

```
#define N 3
long init[N];
seqBool done[N]; /* used in the modified example below */
assign init to {"ss1:init", "ss2:init", "ss3:init"};

state inactive {
  when () {
    for ( i = 0; i < N; i++ ) {
      init[i] = 1;
      pvPut( init[i], ASYNC );
    }
  } state active
}
```

```
state active {
  when ( pvArrayPutComplete( init ) ) {
  } state done

  when ( delay( 10.0 ) ) {
  } state timeout
}
```

pvArrayPutComplete accepts optional arguments to tweak its behaviour. For instance, the following could be inserted before the first transition clause in the active state above. The TRUE argument causes pvPutComplete to return TRUE when any command completed (rather than only when all commands complete). The done argument is the address of a seqBool array of the same size as init ; its elements are set to FALSE for puts that are not yet complete and to TRUE for puts that are complete.

```
when ( pvPutComplete( init, TRUE, done ) ) {
  for ( i = 0; i < N; i++ )
    printf( " %ld", done[i] );
  printf( "\n" );
} state active
```

3.16 Connection Management

All process variable connections are handled by the sequencer via the PV API. Normally the programs are not run until all process variables are connected. However, with the -c compiler flag, execution begins while the connections are being established. The program can test for each variable's connection status with the pvConnected routine, or it can test for all variables connected with the following comparison (if not using dynamic assignment, see Dynamic Assignment, pvAssignCount will be the same as pvChannelCount):

```
pvConnectCount() == pvAssignCount()
```

These routines are described in *Built-in Functions*. If a variable disconnects or re-connects during execution of a program, the sequencer updates the connection status appropriately; this can be tested in a transition clause, as in:

```
when (pvConnectCount() < pvAssignCount()) {
} state disconnected
```

When using dynamic assignment, you should wait for the newly assigned variables to connect, as in:

```
when (pvConnectCount() == pvAssignCount()) {
} state connected

when (delay(10)) {
} state connect_timeout
```

Note that the connection callback may be delivered before or after the initial monitor callback (the PV API does not specify the behavior, although the underlying message system may do so). If this matters to you, you should synchronize the value with an event flag and wait for the event flag to be set before proceeding. See Synchronizing State Sets with Event Flags for an example.

3.17 Multiple Instances and Reentrant Object Code

Occasionally you will create a program that can be used in multiple instances. If these instances run in separate address spaces, there is no problem. However, if more than one instance must be executed simultaneously in a single address space, then the objects must be made reentrant using the +r compiler flag. With this flag all variables are allocated dynamically at run time; otherwise they are declared static. With the +r flag all variables become elements of a common data structure, and therefore access to variables is slightly less efficient.

3.18 Process Variable Element Count

All requests for process variables that are arrays assume the array size for the element count. However, if the process variable has a smaller count than the array size, the smaller number is used for all requests. This count is available with the `pvCount` function. The following example illustrates this:

```
float wf[2000];
assign wf to "{unit}:CavField.FVAL";
int LthWF;
...
LthWF = pvCount(wf);
for (i = 0; i < LthWF; i++) {
  ...
}
pvPut(wf);
...
```

3.19 What's Happening at Run Time

At run time the sequencer blocks until something "interesting" occurs, where "interesting" means things like receiving a monitor from a PV used in a `transition` clause, an event flag changing state, or a delay timer expiring. See section `transitions` in the *SNL Reference for Version 2.2* for a detailed list.

The sequencer then scans the list of `transition` statements for the current state and evaluates each expression in turn. If a `transition` expression evaluates to non-zero the actions within that `transition` block are executed and the sequencer enters the state specified by that `transition` statement. The sequencer then blocks again waiting for something "interesting" to happen.

Note, however, that whenever a new state is entered, the corresponding `transition` conditions for that state are evaluated once without first waiting for events.

3.20 Safe Mode

New in version 2.1.

SNL code can be interpreted in *safe mode*. This must be enabled with the `+s` option, because it changes the way variables are handled and is thus not fully backwards compatible. It should, however, be easy to adapt existing programs to safe mode by making communication between state sets explicit. New programs should no longer use the traditional unsafe mode.

3.20.1 Rationale

In the traditional (unsafe) mode, variables are *not* protected against access from concurrently running threads. Concurrent access to SNL variables was introduced in version 2.0, when implementation of the PV layer switched from the old single threaded CA mode ("preemptive callbacks disabled") to the multi-threaded mode ("preemptive callbacks enabled") in order to support more than one state set per program. This could result in data corruption for variables that are not read and written atomically, the details of which are architecture and compiler dependent (i.e. plain `int` is typically atomic, whereas double is problematic on some, string and arrays on almost all architectures/compilers). Even for plain `int` variables, read-modify-write cycles (like `v++`) cannot be guaranteed to have any consistent result. Furthermore, `conditions` that have been met inside a `transition` clause cannot be relied upon to still hold inside the associated action block.

Concurrent access to SNL variables happens when

- multiple state sets access the same variable, or
- variables are updated from the PV layer due to monitors and asynchronous get operations.

While it is possible to avoid the first case by careful coding (using e.g. event flags for synchronization) it is not possible to guard against the second case as these events can interrupt action statements at any time.

One of the reasons SNL programs have mostly worked in spite of this is that due to the standard CA thread priorities the callback thread does not interrupt the state set threads. Furthermore (and contrary to what many people believe) the VxWorks scheduler does not normally serve threads with equal priority in a round-robin (time-sliced) fashion; instead each thread keeps running until it gets interrupted by a higher priority thread or until it blocks on a semaphore.

However, RTEMS does time-share threads at the same priority, while Linux and Windows may or may not honor thread priorities, depending on the system configuration. Most importantly, priorities should only be used to improve latency for certain operations (at the cost of others) and never should be relied upon for program correctness.

Safe mode solves all these problems by changing the way variables, particularly global variables, are interpreted.

3.20.2 How it Works

In safe mode, all variables –except event flags– are interpreted as if they were *local to the state set*. This means that setting a variable (even a global variable) in one state set does *not* automatically change its value as seen by other state sets. State sets are effectively isolated against each other, and all communication between them must be explicit. They are also isolated against updates by callbacks from the PV layer except at those points where they don't do anything i.e. when they wait for events in a `transition` clause. In safe mode, variable values get updated right before the `condition`s are evaluated, or when explicitly calling synchronization functions like `pvGetComplete` or `pvGet` (the latter only if called in synchronous mode), as well as `efTest` and `efTestAndClear`. The documentation for the built-in functions explains the details.

For instance, with the declaration

```
int var;
assign var;
```

the action statement

```
pvPut(var)
```

makes the value of `var` available to other state sets. They will, however, not see the new value until they issue either a (synchronous) `pvGet`, or the variable is declared as monitored and state change `condition`s are evaluated.

The action

```
pvGet(var, SYNC)
```

updates `var` immediately with whatever has been written to it previously via `pvPut` by some other state set. Whereas

```
pvGet(var, ASYNC)
```

has no immediate effect on the variable `var`. Instead, `var` will be updated only if the code calls `pvGetComplete` (and it returns `true`).

Note: This behaviour is exactly the same as with external PVs.

Note: Using `SYNC` or `ASYNC` with anonymous PVs is not very useful since all operations complete immediately.

3.21 Common Pitfalls and Misconceptions

3.21.1 The delay function does not block

A common misconception among new SNL programmers is that the sequencer somehow blocks inside the `delay` function within `transition` statements. This interpretation of the `delay` function is incorrect but understandable given the name. The `delay` function does not block at all, it merely compares its argument with a timer that is reset whenever the state is entered (from the same or another state), and then returns the result (a boolean value). Any blocking (in case the returned value is FALSE and no other condition fires) is done outside of the `delay` function by the run time system. You might want to think of the operation as `elapsed(s)` rather than `delay(s)`.

If your action statements have any sort of polling loops or calls to `epicsThreadSleep` you should reconsider your design. The presence of such operations is a strong indication that you're not using the sequencer as intended.

3.21.2 Using pvPut and monitor in the same state set

Let's say you have a channel variable x that is monitored, and this code fragment:

```
state one {
  when () {
    x = 1;
    pvPut(x);
    x++;
  } state two
}
state two {
  when (x > 1) {
    do_something();
  }
  when (x <= 1) {
    do_something_else();
  }
}
```

This pattern is hazardous in a number of ways. What exactly happens here depends on whether you are using *Safe Mode* or not.

Assuming traditional (unsafe) mode, it is unpredictable which branch in state `two` will be taken. The `pvPut(x)` might cause a monitor event to be posted by the PV that was assigned to `x`. This event will change `x` back to `1` whenever it arrives. This might happen at any time in between the `pvPut(x)` and the testing of the conditions. It could even interrupt in the middle of the `x++` operation. As a result, this code behaves in conpletely unpredictable ways, depending on the timing of the pvPut-monitor round-trip.

In *Safe Mode* things are slightly better: the only point where the event can lead to an update of the variable is *right before* evaluation of the conditions. However, it is still undetermined which branch will be taken.

You might be tempted to test your code and find that "it works", in the sense that the behavior you see appears to consistently chose one of the two branches, perhaps after adding some `delay`s to the conditions. But this impression is **misleading**, since what actually happens depends on details of thread scheduling and priorities and a host of other timing factors, some of which are very hard to control such as network or system load.

If you cannot avoid using pvPut for a monitored variable, then you should at least

1. use *Safe Mode*, *and*
2. either
   (a) make sure any change you make to the variable gets published (using `pvPut`) before you leave the current action block, *or*
   (b) refrain from changing it, and instead copy the value to some other variable and change *that*.

CHAPTER

FOUR

COMPILING SNL PROGRAMS

4.1 The SNL to C Compiler

The SNL to C compiler `snc` compiles the state notation language into C code. The resulting file can then be compiled with a C compiler.

4.1.1 Compiler Options

SNC options start by a plus or minus sign, followed by a single character. A plus sign turns the option on, and a minus turns the option off, unless the option takes an argument (currently only `-o`).

Option	Description
+a	Asynchronous `pvGet`: the program continues without waiting for completion of the `pvGet` operation.
-a	Synchronous `pvGet`: the program waits for completion. This is the default if an option is not specified.
+c	Wait for process variables to connect before allowing the program to begin execution. This is the default.
-c	Allow the program to begin execution before connections are established to all channel.
+d	Turn on run-time debug messages.
-d	Turn off run-time debug messages. This is the default.
+e	Use the new event flag mode. This is the default.
-e	Use the old event flag mode (clear flags after executing a when statement).
+i	Generate registrar procedure that registers shell commands and programs with an IOC shell. This is the default.
-i	Do not generate registrar procedure.
+l	Add line markers to the generated code, so that C compiler messages refer to the SNL source file. This is the default.
-l	Do not produce line markers.
+m	Include main procedure (seqMain.c) for a stand-alone program.
-m	Do not include seqMain.c. This is the default.
+r	Make the generated code reentrant, thus allowing more than one instance of the program to run on an IOC.
-r	Generated code is not reentrant. This is the default.
+s	*Safe Mode*: variables are local to state set and must be communicated explicitly. Implies +r.
-s	Traditional (non-safe) mode. This is the default for compatibility.
+w	Display warning messages. This is the default.
-w	Suppress warnings.
-o	To change the name of the generated C file. Requires an argument.

New in version 2.2.

Option	Description
+W	Display extra warnings for undefined objects.
-W	Suppress extra warnings. This is the default.

Note that `+a` and `-a` are ignored for calls to `pvGet` that explicitly specify `SYNC` or `ASYNC` in the 2nd argument.

Options may also be set from within the program (somewhere between the program name/parameter and the state set definitions), see *Option* in the *SNL Reference for Version 2.2*.

Prior to Version 1.8 of the sequencer, event flags were cleared after a `when` statement executed. Currently, event flags must be cleared explicitly with either `efTestAndClear` or `efClear`. The `-e` compiler option can be used to restore the old behaviour.

4.1.2 Output File

The output file name is that of the input file with the extension replaced with *.c*. The `-o` option can be used to override the output file name.

Actually the rules are a little more complex than stated above: *.st* and single-character extensions are replaced with *.c*; otherwise *.c* is appended to the full file name. In all cases, the `-o` compiler option overrides.

4.1.3 Errors

If `snc` detects an error, it displays a message describing the error and the location in the source file and aborts further compilation. Note, however, that `snc` does *not* contain a type checker: all it knows (and cares) about C is the syntax. This means that many errors will only be found only during the C compilation phase. The C compiler will attributed these to the corresponding location in the SNL source file, since by default `snc` generates line markers in the output that point back to the original source. This can be turned off with the `-l` ("ell") compiler switch.

4.1.4 Warnings

In certain cases `snc` cannot ultimately decide whether the code is erroneous. In such cases it will issue a warning message and continue.

The most prominent example is the use of a variable or CPP macro that has not been declared in the SNL code, but could well be defined when compiling the generated C code (for example if the declaration has been in embedded C code, which `snc` does not interpret at all). Warnings can be suppressed with the `-w` compiler option.

Note that since version 2.1 you can avoid these warnings by declaring such variables in SNL, see the *Foreign Declarations* declaration.

4.2 C Pre-processor

Depending on the application, it might be useful to pre-process the SNL source with a C pre-processor (`cpp`). Using the C pre-processor allows you to include other SNL files, define macros, and perform conditional compilation. `snc` supports this by interpreting `cpp`-generated line markers, so that error and warning messages refer to the line numbers in the un-pre-processed SNL source.

4.3 Complete Build

The C code generated by `snc` from an SNL program is not a complete program, but merely a collection of procedures, data types, and variables. The generated procedures are supposed to be called by the sequencer

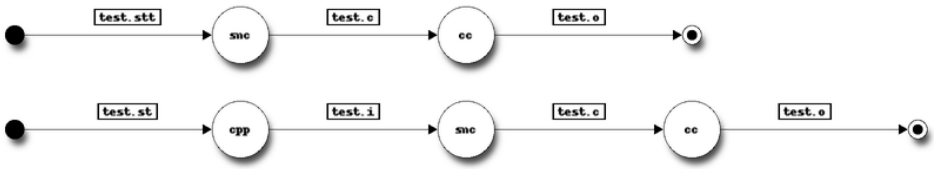
library, which must be linked to the program. Furthermore, the generated code includes a number of header files, both from the sequencer and from EPICS base. Thus the compiler needs to have the EPICS base and sequencer include directories in its include path, and when linking it needs to link with sequencer and some EPICS base libraries.

Assume you have a SNL program file named *test.st* then the steps to build are, in principle:

```
snc test.st
cc -c test.c -o test.o ...additional compiler options...
cc test.o -o test ...additional linker options...
```

or, if SNL sources are to be pre-processed:

```
cc -E -x c test.st > test.i
snc test.i
cc -c test.c -o test.o ...additional compiler options...
cc test.o -o test ...additional linker options...
```



Assuming that `EPICS_BASE` contains the path to your EPICS base installation, and `SEQ` to your sequencer installation, then the extra include directories are:

- $SEQ/include
- $EPICS_BASE/include/os/<osclass>
- $EPICS_BASE/include

where `<osclass>` corresponds to your operating system, see Section 4.3.6 "Specifying osclass specific definitions" of the EPICS Application Developer's Guide. Here is an excerpt:

For vxWorks-* targets <osclass> is vxWorks.
For RTEMS-* targets <osclass> is RTEMS.
For solaris-* targets <osclass> is solaris.
For win32-* targets <osclass> is WIN32.
For linux-* targets <osclass> is Linux.
For hpux-* targets <osclass> is hpux.
For darwin-* targets <osclass> is Darwin.
For aix-* targets <osclass> is AIX.

What libraries to link against and where to find them depends on whether you want to create a stand-alone program (e.g. a Unix executable), or create a library for an IOC.

4.4 Building a Stand-alone Program

The `+m` compiler option can be used to create a stand-alone program, otherwise an iocshell is needed to start sequencer programs. Since version 2.1 the main procedure is no longer hard-coded. Instead, the code generator adds a define and an include statement

```
#define PROG_NAME name_of_your_snl_program
#include "seqMain.c"
```

at the end of the generated C file, where `name_of_your_snl_program` is the name (identifier) whose address is to be passed to the `seq` function. This means you can provide your own version of main simply by placing a file named `seqMain.c` in your source directory (the EPICS build system usually takes care that the source directory is at the front of the C compiler's include path).

A simple default `seqMain.c` is provided and installed into the sequencer's `include` directory. Note that since version 2.1 the default main starts an ioc shell; this can be disabled by providing a `-S` (capital 's') argument. The old `-s` switch is accepted for backward compatibility but does nothing.

4.5 Using makeBaseApp

The easiest way to build a sequencer program is to use `makeBaseApp.pl`, a perl script that comes with your EPICS base. Assuming you have it in your `PATH`, create an empty directory, go there, and issue the command:

```
ben@sarun[1]: .../tmp/test > makeBaseApp.pl -t example
Name the application(s) to be created.
Names given will have "App" appended to them.
Application names? test
ben@sarun[1]: .../tmp/test > ls -l
total 12
-rw-rw-r-- 1 ben ben  467 May 14 22:25 Makefile
drwxrwxr-x 2 ben ben 4096 May 14 22:25 configure
drwxrwxr-x 4 ben ben 4096 May 14 22:25 testApp
```

In `testApp/src/` you will find example `.st` and `.stt` files and a `Makefile` that shows how to define the make variables so that everything is compiled and linked correctly.

All that's left to do is add:

```
SNCSEQ=/path/to/your/seq/installation
```

to `configure/RELEASE` (that is, the one in the `configure` directory that `makeBaseApp.pl` just created).

CHAPTER

FIVE

RUNNING SNL PROGRAMS

This chapter is about how to run and customize a program, once it has been compiled. It also explains how to get information about a running program instance and how to stop a program.

Our running example is the "demo" program which you find under the path `examples/demo` in the source tree.

5.1 Command Syntax

The commands discussed in this chapter can be invoked in different ways. One way is to call them directly from C code. Prototypes for the corresponding C functions can be found in the header file "seqCom.h" which a successful build installs into the "include" directory.

More typical, however, is to call them from some kind of command shell, such as the EPICS IOC shell, or the VxWorks C shell, or the RTEMS cexp shell. In each case the syntax is slightly different, but this is not the place to discuss all these (sometimes subtle) differences. Thus, for simplicity, I will mostly use the IOC shell syntax, indicated by a leading "epics> " shell prompt (so this does not belong to the command itself).

Details about the commands are listed in the *Shell Command Reference* at the end of this chapter.

5.2 Starting a Program

To start an SNL program, the `seq` function is called with three parameters:

1. The address of the `seqProgram` structure. This is a value that is generated by the SNL compiler; it has the same name as the identifier after the `program` keyword.
2. Optionally a string containing parameter definitions, see next section.
3. Optionally a stack size. If `0` (zero), a reasonable default for the target platform is used (`epicsThreadStackSmall`).

For instance, the following command starts the demo program with no parameters and teh default stack size:

```
epics> seq demo
```

Note that the IOC shell defaults missing arguments to zero. Also note that the program gets passed as a string when using the IOC shell, whereas in C you would have to call it like:

```
seq(&demo,0,0);
```

If parameters are to be specified (see next section), this would instead look like

```
epics> seq demo "debug=0,prefix=demo"
```

The sequencer responds with the following output:

```
Sequencer release 2.1.14, compiled Wed Sep 25 12:18:24 2013
Spawning sequencer program "demo", thread 0x98fe120: "demo"
```

The most useful information here is the (EPICS) thread ID `0x98fe120`, since this can be used later on to identify the running program.

BTW, all shell commands write their output to `stdout`.

5.3 Program Parameters

Parameters are a way to customize an SNL program. A program parameter has a name (a string) and a value (also a string). The parameter name should be a valid SNL identifier, while the value can be anything.

Parameters are specified (or defined) in two places: when invoking a program for execution, using the `seq` procedure, or from inside a program after the initial `program` clause. Parameters specified on invocation override those specified in the program. In both cases, the syntax of the actual specification is, roughly:

```
"name1=value1,name2=value2,..."
```

Note that the whole specification is embedded in a single string. A single definition consists of the parameter name, followed by an equal sign, followed by the value. Multiple parameter definitions are separated by comma.

The value is not allowed to contain a comma, or spaces; leading or trailing spaces around the value are ignored. Normal C/SNL string character escaping is admissable, e.g. to embed double quotes or line breaks, but cannot be used to circumvent these limitations.

5.3.1 Special Parameters

The following built-in parameters have special meaning to the sequencer.

`debug = <level>`

This is currently only used by the PV subsystem. A level of 1 or greater turns on debug messages.

`name = <thread_name>`

Normally the thread names are derived from the program name. This parameter specifies an alternative base name for the state set threads.

`priority = <task_priority>`

This parameter specifies initial thread priorities. The value should be an integer between 0 (lowest) and 99 (highest) and will be passed epicsThreadCreate when teh state set threads are created.

`stack = <stack_size>`

This parameter specifies the stack size in bytes. The default is whatever `epicsThreadGetStackSize(epicsThreadStackSmall)` returns.

5.3.2 Using Parameters

Parameter values are available inside the program via the built-in procedure `macValueGet`. They are also (automatically) available for parameter expansion when specifying PV names in `assign` clauses.

5.4 Examining a Program

You can get information about your program at runtime by calling one of several shell commands. They all take a thread ID as first argument to uniquely identify the a running program instance. If this argument is zero (i.e. omitted in the IOC shell), they default to displaying a table that lists the thread IDs of all state sets of all running program instances.

Here is what you might get for the demo program:

```
epics> seqShow
Program Name        Thread ID           Thread Name         SS Name
------------        ---------           -----------         -------
demo                0x88c1da8           demo                light
                    0xb6a1e160          demo_1              ramp
                    0xb6a1e2b8          demo_2              limit
```

When the `seqShow` command is called with one of the thread IDs listed in the table it will give you more detailed information about the running program.

You can get detailed information about the process variables associated with a program by calling `seqChanShow` with a valid thread ID as the first argument. Similarly, `seqChanShow` displays information about monitor queues. Here are some example invocations:

```
epics> seqChanShow 0x88c1da8 demo:lightOn
epics> seqChanShow 0x88c1da8 -
epics> seqChanShow 0x88c1da8 +
epics> seqQueueShow 0x88c1da8
```

The optional second parameter to seqChanShow or seqQueueShow is a channel name, or − to show only those channels which are disconnected, or + to show only those channels which are connected. Both will prompt for input after showing the first (or the specified) channel: hit Enter or a signed number to view more channels or queues; enter `q` to quit.

5.4.1 Stopping a Program

In order to cleanly shut down a running program, use the `seqStop` command:

```
epics> seqStop 0x88c1da8
```

A program can also terminate itself, see `transitions`.

5.5 Shell Command Reference

These are commands to be issued from the IOC shell or VxWorks shell. They can also be called from C (and therefore SNL) code.

Some of these routines behave slightly different depending on whether run under iocsh or a VxWorks shell. This mostly concerns the `threadID` argument: under iocsh, this can in fact be a *thread name* instead of a thread ID. Note, however, that this is unreliable if you have more than one instance of the same program running, since the thread names are identical for all instances. The VxWorks shell version directly takes an epicsThreadID argument and thus does not recognize thread names.

void **seq**(seqProgram **program*, const char **paramdefs*, unsigned *stacksize*)

Start the given program with the given set of parameter definitions and stack size. If `'stacksize` is zero or is omitted, then use a default (EPICS "small" stack). If `paramdefs` is zero or the last two arguments are omitted, then no parameters are defined. Otherwise `paramdefs` should be a string that specifies program parameters as detailed in *Program Parameters*. See also `program_param`.

void **seqShow**()
void **seqShow**(epicsThreadId *threadID*)

The first form shows a table of all programs, program instances, and state sets, e.g.

```
epics> seqShow
Program Name        Thread ID           Thread Name         SS Name
------------        ---------           -----------         -------
demo                0x807e628           demo                light
                    0x809fbc8           demo_1              ramp
```

```
                    0x809fcd8           demo_2              limit
------------        ---------           -----------         -------
demo                0x807fd98           demo                light
                    0xb7100470          demo_1              ramp
                    0xb7100580          demo_2              limit
------------        ---------           -----------         -------
demo                0x80814e0           demo                light
                    0x809fe68           demo_1              ramp
                    0x809ff78           demo_2              limit
```

Note that in this example we have three running instances of a single program named 'demo', each of which consists of three state sets running in its own thread.

The second form displays the internal state of a running program instance. The threadID parameter must be one of the program's state set thread IDs as listed in the above table.

For instance, for the above example we might get

```
epics> seqShow 0x807e628
State Program: "demo"
  thread priority = 50
  number of state sets = 3
  number of syncQ queues = 0
  number of channels = 6
  number of channels assigned = 6
  number of channels connected = 6
  number of channels monitored = 5
  options: async=0, debug=0, newef=1, reent=1, conn=1, main=0
  user variables: address = 0x807d158, length = 44

  State Set: "light"
  thread name = demo;  Thread id = 0x807e628
  First state = "START"
  Current state = "LIGHT_OFF"
  Previous state = "START"
  Elapsed time since state was entered = 3.0 seconds
  Get in progress = [000000]
  Put in progress = [000000]
  Queued time delays:

  State Set: "ramp"
  thread name = demo_1;  Thread id = 0x809fbc8
  First state = "START"
  Current state = "RAMP_UP"
  Previous state = "RAMP_UP"
  Elapsed time since state was entered = 0.1 seconds
  Get in progress = [000000]
  Put in progress = [000000]
  Queued time delays:
        delay[0]=0.100000

  State Set: "limit"
  thread name = demo_2;  Thread id = 0x809fcd8
  First state = "START"
  Current state = "START"
  Previous state = ""
  Elapsed time since state was entered = 3.0 seconds
  Get in progress = [000000]
  Put in progress = [000000]
  Queued time delays:
```

void **seqChanShow** (epicsThreadId *threadID*)
void **seqChanShow** (epicsThreadId *threadID*, const char *)

Display channel information for the program instance specified by the given threadID. If a second argument is given, it is interpreted as part of a channel name. Only channels whose name contains the given string as substring are displayed. The name can be preceded by a single "-" or "+" sign, signifying that only disconnected ("-") or connected ("+") channels should be displayed.

The procedure displays one channel at a time, starting with the first matching one, and then asks the user for input. This input can be

a (signed) integer: increase / decrease current channel number by the given amount, then display the current channel

minus sign ("-"): same as "-1"

plus sign ("+"), empty string (return): same as "+1"

anything else: quit, i.e. back to the shell

If user interaction causes the channel number to leave the range (i.e. less than zero, greater or equal to number of channels), the command quits, too.

void **seqQueueShow** (epicsThreadId *threadID*)

Display information about queued channels. For example

```
epics> seqShow
Program Name        Thread ID           Thread Name         SS Name
------------        ---------           -----------         -------
syncqTest           0x8053e60           syncqTest           get
                    (nil)               (no thread)         get1
                    (nil)               (no thread)         put
                    (nil)               (no thread)         flush
epics> seqQueueShow 0x8053e60
State Program: "syncqTest"
Number of queues = 2
  Queue #0: numElems=5, used=0, elemSize=136
Next? (+/- skip count, q=quit)

  Queue #1: numElems=5, used=0, elemSize=56
Next? (+/- skip count, q=quit)
```

The command is interactive and accepts the same inputs as seqChanShow.

void **seqcar** (int *level*)

The name stands for "sequencer channel access report". It displays channel connection information. If level <= 1, or no level argument is given, only a summary line is displayed, for example

```
Total programs=3, channels=18, connected=18, disconnected=0
```

For level > 1, connection information is displayed for all channels of all running programs. For instance

```
epics> seqcar 2
  Program "demo"
    Variable "light" connected to PV "demo1:light"
    Variable "lightOn" connected to PV "demo1:lightOn"
    Variable "lightOff" connected to PV "demo1:lightOff"
    Variable "voltage" connected to PV "demo1:voltage"
    Variable "loLimit" connected to PV "demo1:loLimit"
    Variable "hiLimit" connected to PV "demo1:hiLimit"
  Program "demo"
    Variable "light" connected to PV "demo2:light"
    Variable "lightOn" connected to PV "demo2:lightOn"
    Variable "lightOff" connected to PV "demo2:lightOff"
    Variable "voltage" connected to PV "demo2:voltage"
    Variable "loLimit" connected to PV "demo2:loLimit"
    Variable "hiLimit" connected to PV "demo2:hiLimit"
  Program "demo"
```

```
    Variable "light" connected to PV "demo3:light"
    Variable "lightOn" connected to PV "demo3:lightOn"
    Variable "lightOff" connected to PV "demo3:lightOff"
    Variable "voltage" connected to PV "demo3:voltage"
    Variable "loLimit" connected to PV "demo3:loLimit"
    Variable "hiLimit" connected to PV "demo3:hiLimit"
Total programs=3, channels=18, connected=18, disconnected=0
```

void **seqStop** (epicsThreadId *threadID*)

Initiate a clean program exit. Running state `transitions` are completed, then all state set threads exit, all channels are disconnected, and finally allocated resources are freed.

CHAPTER

SIX

ESCAPE TO C CODE

Because the SNL does not support the full C language, C code may be escaped in the program. The escaped code is not compiled by snc, instead it is literally copied to the generated C code. There are two escape methods:

1. Any code between %% and the next newline character is escaped. Example:

   ```
   %% for (i=0; i < NVAL; i++)
   ```

2. Any code between %{ and }% is escaped. Example:

   ```
   %{
   extern float smooth();
   extern LOGICAL accelerator_mode;
   }%
   ```

   Note that text appearing on the same line after %{ and before }% also belongs to the literal code block.

A variable or preprocessor macro declared in escaped C code is foreign to the SNL, just as if it were declared in C code extern to the SNL program and its use will give a warning if no foreign declaration preceeds it.

6.1 Preprocessor Directives

A very common pitfall is the use of preprocessor directives, such as #include, in multi-line literal C code blocks in conjunction with using the preprocessor on the unprocessed SNL code (which happens by the default with the standard build rules if the name of the source file has the .st extension).

For instance with

```
%{
#include <abcLib.h>
/* ... */
}%
```

the header file will be included *before* the SNL compiler parses the program, which is most probably not what you wanted to happen here. For instance, if the header contains macro definitions, these will not be in effect when the rest of the C code block gets compiled.

You can defer interpretation of a preprocessor directive until after snc has compiled the code to C, by ensuring that some extra non-blank characters appear in front of the # sign, so cpp does not recognize the directive. For instance

```
%%#include <abcLib.h>
```

or

```
%{#include <abcLib.h>}%
```

6.2 Defining C Functions within the Program

Escaped C code can appear in many places in the SNL program. But only code that appears at the top level, i.e. outside any SNL code block will be placed at the C top level. So, C *function definitions* must be placed there; this means either inside the definitions section (`initial_defns`) or at the end of the program. For example:

```
program example
...
/* last SNL statement */
%{
    static float smooth (pArray, numElem)
    { ... }
}%
```

It matters where the function is defined: if it appears at the end of the program, it can see all the variable and type definitions generated by the SNL compiler, so if your C function accesses a global SNL variable, you must place its definition at the end. However, this means that the generated code for the SNL program does not see the C function definition, so you might want to place a separate prototype for the function in the definitions section (i.e. before the state sets).

Remember that you can *call* any C function from anywhere in the SNL program without escaping. You can also link the SNL program to C objects or libraries, so the only reason to put C function definitions inside the SNL program is if your function accesses global SNL variables.

6.3 Calling PV Functions from C

The built-in SNL functions such as `pvGet` cannot be directly used in user-supplied functions. However, most of the built-in functions have a C language equivalent with the same name, except that the prefix `seq_` is added (e.g. `pvGet` becomes `seq_pvGet`). These C functions expect an additional first argument identifying the calling state set, which is available in action code under the name `ssId`. If a process variable name is required, the index of that variable must be supplied. This index is obtained via the `pvIndex` function (which must be called from SNL code, not C code, to work).

If the program is compiled with the `+r` option, user functions cannot directly access SNL variables, not even global ones. Instead, variables (or their address) should be passed to user functions as arguments. Alternatively, you can pass the `pVar` variable of type `USER_VAR*` and access SNL variables as structure members of `pVar`. See next section for details and an example.

The prototypes for the C functions corresponding to the built-in SNL functions as well as additional supporting macros and type definitions can be found in the header file `seqCom.h`. This header file is always included by the generated C code.

6.4 Variable Modification for Reentrant Option

If the reentrant option (`+r`) is specified to `snc` then all variables are made part of a structure. Suppose we have the following top-level declarations in the SNL program:

```
int sw1;
float v5;
short wf2[1024];
```

The C file will contain the following declaration:

```
struct UserVar {
    int sw1;
    float v5;
    short wf2[1024];
};
```

The sequencer allocates the structure area at run time and passes a pointer to this structure into the program. This structure has the following type:

```
struct UserVar *pVar;
```

Reference to variable `sw1` is made as

```
pVar->sw1
```

This conversion is automatically performed by `snc` for all SNL statements, but you will have to handle escaped C code yourself.

Note: *Safe Mode* (enabled with the `+s` option) implies reentrant mode. In safe mode, each state set has its own copy of the `UserVar` struct. You can operate on its members in any way you like, including taking the address of variables and passing them to C functions.

Here is a stupid example of a C function that does a pvGet, increments the variable, and then does a pvPut:

```
program userfunc

option +r;

%{
static void incr(SS_ID ssId, int *pv, VAR_ID v)
{
    seq_pvGet(ssId, v, SYNC);
    *pv += 1;
    seq_pvPut(ssId, v, SYNC);
}
}%

int i;
assign i to "counter";

foreign ssId;

ss myss {
    state doit {
        when (delay(1)) {
            incr(ssId, &i, pvIndex(i));
        } state doit
    }
}
```

CHAPTER

SEVEN

SNL REFERENCE FOR VERSION 2.2

This chapter gives a detailed reference for the SNL syntax and semantics and for the built-in functions of SNL in version 2.2. The documentation for version 2.1 is and will remain available, too.

Formal syntax is given in BNF. Multiple rules for the same nonterminal symbol mean that any of the given rules may apply. Terminal symbols are enclosed in double quotes.

7.1 Lexical Syntax

The lexical syntax is not specified formally in this document, since it is almost identical to that of C. There are only two exceptions:

- Embedded C Code, and
- more reserved words.

We use an informal notation with explanatory text in agle brackets standing in for the formal definition.

7.1.1 Comments

```
comment  ::=   "/*" <anything> "*/"
comment  ::=   "//" <anything until end of line>
```

C-style comments may be placed anywhere in the program. They are treated as white space. As in C, comments cannot be nested.

New in version 2.2.

C++ style comments are allowed, too.

7.1.2 Identifiers

```
identifier  ::=   <same as in C>
```

Identifiers follow the same rules as in C. They are used for variables (including foreign variables and event flags), the program name, states, state sets, and options.

7.1.3 Literals

```
integer_literal         ::=   <same as in C>
```

```
floating_point_literal  ::=   <same as in C>
string_literal          ::=   <same as in C>
```

The lexical syntax of identifiers, as well as numeric and string literals is exactly as in C, including automatic string concatenation, character literals, and octal, decimal, and hexadecimal integer literals.

7.1.4 Embedded C Code

```
embedded_c_code  ::=   "%{" <anything> "}%"
embedded_c_code  ::=   "%%" <anything> "\n"
```

A sequence of characters enclosed between "%{" and "}%" is used literally and without further parsing as if it were a complete declaration or statement, depending on where it appears.

A sequence of characters enclosed between "%%" and the next line ending is treated similarly, except that it is stripped of leading and trailing whitespace and inserted in the output with the current indentation.

Note that embedded C code fragments are *not* allowed to appear in expressions.

See *Escape to C Code* for examples and rationale.

Embedded C code fragments are the cause for one of the two conflicts in the grammar. The reason is that the parser cannot always decide whether such a fragment is a declaration or a statement. (The other conflict is due to the infamous "dangling else".)

7.1.5 Line Markers

```
line_marker  ::=   "#" line_number "\n"
line_marker  ::=   "#" line_number file_name "\n"
line_number  ::=   <non-empty sequence of decimals>
file_name    ::=   <like string_literal, without automatic string concatenation>
```

Line markers are interpreted exactly as in C, i.e. they indicate that the following symbols are really located in the given source file (if any) at the given line.

Note: `line_number` may only contain decimal numbers, and `file_name` must be a single string (no automatic string concatenation).

Line markers are typically generated by preprocessors, such as `cpp`.

7.2 Program

```
program  ::=   "program" identifier program_param initial_defns entry state_sets exit fi
```

This is the overall structure of an SNL program. After the keyword "program" comes the name of the program, followed by an optional program parameter, initial global definitions, an optional entry block, the state sets, an optional exit block, and then again global definitions.

7.2.1 Program Name and Parameters

The program name is an identifier. It is used as the name of the global variable which contains or points to all the program data structures (the address of this global variable is passed to the `seq` function when creating the run-time sequencer). It is also used as the base for the state set thread names unless overridden via the *name* parameter (see *Program Parameters*).

```
program_param  ::=   "(" string ")"
program_param  ::=
```

The program name may be followed by an optional string enclosed in parentheses. The string content must be a list of comma-separated parameters in the same form as if specified on invocation (see *Program Parameters*).

Note: Parameters specified on invocation override those specified in the program.

```
initial_defns  ::=   initial_defns initial_defn
initial_defns  ::=
initial_defn   ::=   assign
initial_defn   ::=   monitor
initial_defn   ::=   sync
initial_defn   ::=   syncq
initial_defn   ::=   declaration
initial_defn   ::=   option
initial_defn   ::=   funcdef
initial_defn   ::=   structdef
initial_defn   ::=   c_code
```

```
final_defns  ::=   final_defns final_defn
final_defns  ::=
final_defn   ::=   funcdef
final_defn   ::=   structdef
final_defn   ::=   c_code
```

Global (top-level) definitions, see Definitions for details. Note that in the final definition section only function definitions and literal code blocks are allowed.

7.2.2 Global Entry and Exit Blocks

```
entry  ::=   "entry" block
entry  ::=
exit   ::=   "exit" block
exit   ::=
```

An SNL program may specify optional entry code to run prior to running the state sets, and exit code to run after all state sets have exited. Both are run in the context of the first state set thread.

(Global entry and exit blocks should not be confused with *State Entry and Exit Blocks*, which have the same syntax but are executed at each `transition` from/to a new state.)

The entry or exit code is a regular SNL code block and thus can contain local variable declarations. Global entry code is always executed after initiating connections to (named) PVs, and exit code is executed before connections are shut down. Furthermore, if the global `+c` option is in effect, the entry code is executed only after all channels are connected and monitored channels have received their first monitor event. All built-in PV functions can be

expected to work in global entry and exit blocks.

Note that global entry and exit blocks operate as if executed as part of the first state set. This means that in *Safe Mode* changes to global variables made from inside the entry block are not visible to state sets other than the first unless explicitly communicated (e.g. by calling pvPut). The fact that the first state set plays a special role is a historical accident, rather than conscious design. I might be tempted to redesign this aspect in a future version, for instance by giving entry and exit blocks their own dedicated virtual state set.

7.2.3 Final C Code Block

```
c_code  ::=  embedded_c_code
```

Any number of embedded C code blocks may appear after state sets and the optional exit block. See *Escape to C Code*.

7.3 Definitions

7.3.1 Function Definitions

New in version 2.2.

```
funcdef  ::=  basetype declarator block
```

Function definitions are very much like those in C.

The most important difference is that functions declared in SNL get passed certain implicit parameters. This makes it possible to call built-in functions as if the code were a normal action block. Note, however, that there is currently no way to *pass* channel ("assigned") variables to SNL functions in such a way that you can call built-in functions on them. The "assigned" status of such a variable is lost and it gets passes to the function as a normal C value. This means you can call built-in functions that expect such a variable only if the variable was declared at the top-level. Lifting this limitation is planned for a future release.

Another difference from C is that SNL functions automatically scope over the whole program, no matter where they are defined. It is not necessary (but allowed) to repeat the function declaration (without the defining code block, commonly referred to as a "function prototype") at the top of the program.

7.3.2 Type Definitions

New in version 2.2.

```
structdef     ::=  "struct" identifier members ";"
members       ::=  "{" member_decls "}"
member_decls  ::=  member_decl
member_decls  ::=  member_decls member_decl
member_decl   ::=  basetype declarator ";"
member_decl   ::=  c_code
```

Type definitions are currently limited to struct definitions and may appear only at the top level. They are translated to the equivalent struct definition in C. Differences to C are that bit fields are not supported, and that member types are limited to SNL types (built-in or user defined).

7.3.3 Variable Declarations

```
declaration       ::=   basetype init_declarators ";"
init_declarators  ::=   init_declarator
init_declarators  ::=   init_declarators "," init_declarator
init_declarator   ::=   declarator
init_declarator   ::=   declarator "=" init_expr
declarator        ::=   variable
declarator        ::=   "(" declarator ")"
declarator        ::=   declarator subscript
declarator        ::=   declarator "(" param_decls ")"
declarator        ::=   "*" declarator
declarator        ::=   "const" declarator
variable          ::=   identifier
init_expr         ::=   "{" init_exprs "}"
init_expr         ::=   "(" type_expr ")" "{" init_exprs "}"
init_expr         ::=   expr
init_exprs        ::=   init_exprs "," init_expr
init_exprs        ::=   init_expr
init_exprs        ::=
```

New in version 2.1.

You can declare more than one variable in a single declaration (comma separated) and add pointer and array markers (subscripts) ad libitum as well as initializers.

Changed in version 2.2: Function declarations and initializers with type casts.

As in C, some combinations of type operators are not allowed, for instance because they are redundant (like declaring a function or array const). This is not specified in the SNL grammar. There are some checks that warn about obviously wrong combinations, but as with type checking, most of the work is off-loaded to the C compiler.

The also remain some limitations:

- arrays must have a defined size: the expression inside the subscript brackets is not optional as in C; it must also be an integer literal, not a constant expression as in C.
- you cannot declare new types or type synonyms
- the "volatile" type qualifier is not supported
- neither are storage specifiers ("static", "extern", "auto", register")
- only certain base types are allowed, see *below*)

Some of these restrictions may be lifted in future versions.

const

New in version 2.2.

As in C, declarations may involve the "const" keyword to make parts of the thing declared constant i.e. immutable. The SNL syntax treats "const" as a prefix type operator, exactly like the pointer "*" operator.

Note: This correctly specifies a large subset of the C syntax, but there are some limitations. For instance, in C you can write

```
const int x;   /* attention: invalid in SNL */
```

as an alternative notation for

```
int const x;
```

This is not allowed in SNL to keep the parser simple and orthogonal.

Otherwise, "const" behaves like in C. For instance,

```
char const *p;
```

declares a mutable pointer to constant char (possibly more than one), while

```
char *const p;
```

declares a constant pointer to one or more mutable chars, and

```
char *const *p;
```

a constant pointer to constant chars.

As in C, declarations (and similarly Type Expressions) should be read inside out, starting with the identifier and working outwards, respecting the precedences (postfix binds stronger than prefix). This works for "const" exactly as for array subscripts, pointers, and parameter lists.

Foreign Declarations

New in version 2.1.

Deprecated since version 2.2.

```
declaration  ::=  "foreign" variables ";"
variables    ::=  variable
variables    ::=  variables "," variable
```

Foreign declarations are used to let the SNL compiler know about the existence of C variables or C preprocessor macros (without parameters) that have been defined outside the SNL program or in escaped C code. No warning will be issued if such a variable or macro is used in the program even if warnings are enabled.

Changed in version 2.2.

It is no longer needed to declare struct or union members as foreign entities, since the parser no longer confuses them with variables. See section Postfix Operators below.

Also, warnings for undefined entities are now disabled by default. You can use the "extra warnings" option `+W` to enable them. In contrast to older versions, this will also warn about foreign functions when called from SNL code. Note that enabling extra warnings is normally not needed: if a variable or function is actually undefined in the generated C code, the C compiler will issue a warning (or an error) anyway.

Basic Types

These are the allowed base types, of which you can declare variables, or pointers or arrays etc.

```
basetype  ::=  prim_type
basetype  ::=  "evflag"
```

New in version 2.2.

```
basetype  ::=  "enum" identifier
basetype  ::=  "struct" identifier
```

```
basetype  ::=   "union" identifier
basetype  ::=   "typename" identifier
basetype  ::=   "void"
```

You can use enumerations, structs, unions and typedefs in declarations and type expressions. The "void" type is also allowed, subject to the same restrictions as in C.

Note: Any use of a type alias (defined using "typedef" in C) *must* be preceded by the "typename" keyword. This keyword has been borrowed from C++, but in contrast to C++ using "typename" is *not* optional.

```
prim_type  ::=   "char"
prim_type  ::=   "short"
prim_type  ::=   "int"
prim_type  ::=   "long"
prim_type  ::=   "unsigned" "char"
prim_type  ::=   "unsigned" "short"
prim_type  ::=   "unsigned" "int"
prim_type  ::=   "unsigned" "long"
prim_type  ::=   "int8_t"
prim_type  ::=   "uint8_t"
prim_type  ::=   "int16_t"
prim_type  ::=   "uint16_t"
prim_type  ::=   "int32_t"
prim_type  ::=   "uint32_t"
prim_type  ::=   "float"
prim_type  ::=   "double"
prim_type  ::=   "string"
```

Primitive types are those base types that correspond directly to some (scalar) EPICS type. They have the same semantics as in C, except "string" that does not exist in C. See subsection Strings below.

New in version 2.1.

Since the standard numeric types in C have implementation defined size, fixed size integral types that correspond to the ones in the C99 standard have been added. However, int64_t and uint64_t are not supported because primitive types must correspond to the primitive EPICS types, and EPICS does not yet support 64 bit integers.

Type Expressions

New in version 2.2.

```
type_expr    ::=   basetype
type_expr    ::=   basetype abs_decl
abs_decl     ::=   "(" abs_decl ")"
abs_decl     ::=   subscript
abs_decl     ::=   abs_decl subscript
abs_decl     ::=   "(" param_decls ")"
abs_decl     ::=   abs_decl "(" param_decls ")"
abs_decl     ::=   "*"
abs_decl     ::=   "*" abs_decl
abs_decl     ::=   "const"
abs_decl     ::=   "const" abs_decl
param_decls  ::=
param_decls  ::=   param_decl
param_decls  ::=   param_decls "," param_decl
param_decl   ::=   basetype declarator
```

```
param_decl   ::=   type_expr
```

Type expressions closely follow declaration syntax just as in C. They are used for parameter declarations as well as type casts and the special sizeof operator (see Prefix Operators).

Using "const" in type expressions is subject to the same restrictions as in declarations (see Variable Declarations).

Strings

The type `string` is defined in C as:

```
typedef char string[MAX_STRING_SIZE];
```

where `MAX_STRING_SIZE` is a constant defined in one of the included header files from EPICS base. I know of no EPICS version where it is different from 40, but to be on the safe side I recommend not to rely too much on the numeric value. (You can use `sizeof(string)` in SNL expressions.)

Note: In contrast to C, in SNL `string s` is *not* a synonym for `char s[MAX_STRING_SIZE]`, since variables of type `string` are treated differently when it comes to interacting with PVs: the former gets requested with type DBR_STRING and a count of one, while the latter gets requested with type DBR_CHAR and a count of `MAX_STRING_SIZE`.

Event Flags

Event flags are values of an abstract data type with four operations defined on them: `efSet`, `efClear`, `efTest`, and `efTestAndClear`.

An event flag `e` can act as a binary semaphore, allowing exactly one state set to continue, if `when(efTestAndClear(e))` is used to wait and `efSet(e)` to signal. Event flags can be coupled to changes of a PV using the `sync` clause, so that the flag gets set whenever an event happens.

You cannot declare arrays of or pointers to event flags, since event flags are not translated to C variables in your program.

See *Synchronizing State Sets with Event Flags*.

7.3.4 Variable Scope and Life Time

Variables are statically scoped, i.e. they are visible and accessible to the program only inside the smallest `block` enclosing the declaration; if declared outside of any block, i.e. at the top level, they are visible everywhere in the program.

This is quite similar to C and other statically scoped programming languages. However, there are two differences to C:

First, global variables are always local to the program, similar to variables that have been declared "static" in C.

Second, in C, a variable's life time is defined by its scope: when the scope is left, the variable becomes undefined. That is, life time (also called dynamic scope) follows static scope. This is generally the same in SNL, with two notable *exceptions*: A variable declared local to a state set or local to a state continues to exist *as long as the program runs*, just as if it were declared at the top level, i.e. before any state sets. We say that such variables have *global life time*.

Only variables with global life time can be assigned to a process variable, monitored, synced etc. If they have an initializer, they will be initialized only once when the program starts.

Variables declared local to an action block or a compound statement behave exactly as in C. They can *not* be assigned to a PV; and (if they have an initializer) they become re-initialized each time the block is entered.

The rationale for this is that, while bringing a normal variable into and out of scope is a very cheap operation, establishing a connection to a PV is not.

7.3.5 Variable Initialization

Initializers for variables of global life time (i.e. globals as well as state set and state local ones) must respect the usual C rules for static variable initializers. Particularly, they must be *constant expressions* in the C sense, i.e. calculable at compile time, which also means they must not refer to other variables.

Variables of global life time are initialized before *any* other code is executed.

Initializers for other variables (i.e. those with local lifetime) behave exactly as regular local C variables. Particularly, an initializer for such a variable may refer to other variables that have been declared (and, possibly, initialized) in an outer scope, and also to those that have previously been declared (and, possibly, initialized) in the same scope.

7.3.6 Process Variables

The syntactic constructs in this section allow program variables to be connected to externally defined process variables.

assign

```
assign     ::=   "assign" variable to string ";"
assign     ::=   "assign" variable subscript to string ";"
assign     ::=   "assign" variable to "{" strings "}" ";"
assign     ::=   "assign" variable ";"
to         ::=   "to"
to         ::=
strings    ::=   strings "," string
strings    ::=   string
strings    ::=
subscript  ::=   "[" integer_literal "]"
```

This assigns or connects program variables to named or anonymous process variables.

There are four variants of the `assign` statement. The first one assigns a (scalar or array) variable to a single process variable. The second one assigns a single element of an array variable to a single process variable. The third one assigns elements of an array variable to separate process variables.

For the third variant, if the number of PV names does not match the number of elements in the array, then the rule is that

- missing PV names default to "", and
- excess PV names are discarded.

New in version 2.1.

The fourth variant serves as an abbreviation for the first variant, in the special case where the PV name is empty ("").

Assigned variables must be of global life time, see *Variable Scope and Life Time*. Assigned variables, or separately assigned elements of an array, can be used as argument to built-in `pvXXX` procedures (see Built-in Functions). This is the primary means of interacting with process variables from within an SNL program.

Only certain types of variables can be assigned to a PV: allowed are numeric types (char, short, int, long, and their unsigned variants) and strings (these are sometimes referred to as *scalar types*), as well as one or two dimensional arrays of these.

Process variable names are subject to *parameter expansion*: substrings of the form

```
{parametername}
```

(i.e. an identifier enclosed in curly braces) are expanded to the value of the program parameter *if* a corresponding parameter is defined (either inside the program or as an extra argument on invocation, see `seq`); otherwise no expansion takes place.

If the process variable name (after expansion) is an empty string, then no actual assignment to any process variable is performed, but the variable is marked for potential (dynamic) assignment with `pvAssign`.

Note: An `assign` clause using an empty string for the PV name is interpreted differently in *Safe Mode*, see *Anonymous Channels*.

An array variable assigned wholesale to one process variable (using the first syntactic variant above) or an element of a two-dimensional variable assigned to an array process variable (using the second syntactic variant) will use either the length of the array or the native count for the underlying process variable, whichever is smaller, when communicating with the underlying process variable. The native count is determined when a connection is established. For anonymous PVs, the length of the array is used.

Pointer types may not be assigned to process variables.

monitor

```
monitor        ::=   "monitor" variable opt_subscript ";"
opt_subscript  ::=   subscript
opt_subscript  ::=
```

This sets up a monitor for an assigned variable or array element.

Monitored variables are automatically updated whenever the underlying process variable changes its value. Note, however, that this depends on the configuration of the underlying PV: some PVs post an update event only if the value changes by at least a certain amount. Also, events may be posted, even if no actual change happens, i.e. the value remains the same. The details can be found in the EPICS Record Reference Manual.

sync

```
sync        ::=   "sync" variable opt_subscript to event_flag ";"
event_flag  ::=   identifier
```

This declares a variable to be synchronized with an event flag.

When a monitor is posted on any of the process variables associated with the event flag (and these are `monitor`ed), or when an asynchronous get or put operation completes, the corresponding event flag is set.

The variable must be `assign`ed and `monitor`ed. A variable can be mentioned in at most one sync clause, but an event flag may appear in more than one such clause. The variable may be an array, and as such may be associated with multiple process variables.

Changed in version 2.1.

- It is now allowed to sync an event flag to more than one variable.
- There is now a run-time equivalent to the sync clause, see the built-in function `pvSync`.

syncQ

```
syncq       ::=   "syncq" variable opt_subscript to event_flag syncq_size ";"
syncq       ::=   "syncq" variable opt_subscript syncq_size ";"
```

```
syncq_size  ::=  integer_literal
syncq_size  ::=
```

This declares a variable to be queued.

When a monitor is posted on any of the process variables associated with the given program variable, the new value is written to the end of the queue. If the queue is already full, the last (youngest) entry is overwritten. The `pvGetQ` function reads items from the queue.

The variable must be `assign`ed and `monitor`ed. Specifying a size (number of elements) for the queue is optional. If a size is given, it must be a positive decimal number, denoting the maximum number of elements that can be stored in the queue. A missing size means that it defaults to 100 elements. The variable can be an array, and may be associated with multiple process variables, in which case there is still only one queue, but any change in any of the involved PVs will add an entry to the queue.

New in version 2.1.

You can use "syncq" (all lower case) as keyword instead of "syncQ". The latter may be deprecated in a future version.

Changed in version 2.1.

Not giving a queue size (thus relying on the default of 100 elements) is now *deprecated* and the compiler will issue a warning. The reason for this is that queues are now statically allocated, which can result in a large memory overhead especially if the variable is an array associated with a single PV. (A default queue size of 1 would be much more useful, but for compatibility I kept it at 100 as in previous versions.)

Changed in version 2.1.

A queued variable no longer needs to be associated with an event flag. The first form of the `syncq` clause is now merely an abbreviation for a `sync` clause together with a `syncq` of the second form, i.e.

```
syncq var to ef qsize;
```

is equivalent to

```
sync var to ef;
syncq var qsize;
```

Forcing the association with an event flag was never really necessary, since `pvGetQ` already checks and returns whether the queue is empty or not; and any state set that mentions a variable in a `transition` clause automatically gets woken up whenever the variable changes due to a monitor event. On the other hand, relying on the event flag being set as an indication that the queue is non-empty has always been unreliable since another `pvGetQ` might have intervened and emptied the queue between the two calls.

Note that `pvGetQ` clears an event flag associated with the variable if the queue becomes empty after removing the head element.

7.3.7 Option

```
option        ::=  "option" option_value identifier ";"
option_value  ::=  "+"
option_value  ::=  "-"
```

The option name is any combination of option letters. Multiple options can be clobbered into a single option clause, but only a single option value is allowed. Option value "+" turns the given options on, a"-" turns them off.

Examples:

```
option +r; /* make code reentrant */
option -ca; /* synchronous pvGet and don't wait for channels to connect */
```

Unknown option letters cause a warning to be issued, but are otherwise ignored.

The same syntax is used for global options and state options. The interpretation, however, is different:

Global (top-level) options are interpreted as if the corresponding compiler option had been given on the command line (see *Compiler Options*). Global option definitions take precedence over options given to the compiler on the command line.

State options occur inside the state construct and affect only the state in which they are defined, see State Option.

7.4 State Set

```
state_sets  ::=   state_sets state_set
state_sets  ::=   state_set
state_set   ::=   "ss" identifier "{" ss_defns states "}"
ss_defns    ::=   ss_defns ss_defn
ss_defns    ::=
```

A program contains one or more state sets. Each state set is defined by the keyword "ss", followed by the name of the state set (an identifier). After that comes an opening brace, optionally state set local definitions, a list of states, and then a closing brace.

State set names must be unique in the program.

7.4.1 State Set Local Definition

```
ss_defn  ::=   assign
ss_defn  ::=   monitor
ss_defn  ::=   sync
ss_defn  ::=   syncq
ss_defn  ::=   declaration
```

Inside state sets are allowed variable declarations and process variable definitions (`assign`, `monitor`, `sync`, and `syncq`).

See variable scope for details on what local definitions mean.

7.4.2 State

```
states       ::=   states state
states       ::=   state
state        ::=   "state" identifier "{" state_defns entry transitions exit "}"
state_defns  ::=   state_defns state_defn
state_defns  ::=
```

A state set contains one or more states. Each state is defined by the keyword "state", followed by the name of the state (and identifier), followed by an opening brace, optionally state local definitions, an optional entry block, a list of `transitions`, an optional exit block, and finally a closing brace.

State names must be unique in the sate set to which they belong.

State Local Definition

```
state_defn  ::=  assign
state_defn  ::=  monitor
state_defn  ::=  sync
state_defn  ::=  syncq
state_defn  ::=  declaration
state_defn  ::=  option
```

State Option

The syntax for a state option is the same as for global options (see *Option*).

The state options are:

+t	Reset delay timers each time the state is entered, even if entered from the same state. This is the default.
-t	Don't reset delay timers when entering from the same state. In other words, the `delay` function will return whether the specified time has elapsed from the moment the current state was entered from a different state, rather than from when it was entered for the current iteration.
+e	Execute `entry` blocks only if the previous state was not the same as the current state. This is the default.
-e	Execute `entry` blocks even if the previous state was the same as the current state.
+x	Execute `exit` blocks only if the next state is not the same as the current state. This is the default.
-x	Execute `exit` blocks even if the next state is the same as the current state.

For example:

```
state low {
   option -e; /* Do entry{} every time ... */
   option +x; /* but only do exit{} when really leaving */
   entry { ... }
   ...when ()...
   exit { ... }
}
```

State Entry and Exit Blocks

The syntax is the same as for Global Entry and Exit Blocks.

Entry blocks are executed when the state is entered, before any of the `condition`s for state `transitions` are evaluated.

Exit blocks are executed when the state is left, after the `transition` block that determines the next state.

Transitions

```
transitions  ::=  transitions transition
transitions  ::=  transition
transition   ::=  "when" "(" condition ")" block "state" identifier
condition    ::=  opt_expr
```

A state transition starts with the keyword "when", followed by a condition (in parentheses), followed by a block,

and finally the keyword "state" and then the name of the target state (which must be a state of the same state set).

The condition must be a valid boolean expression (see Expressions). If there is no condition given, it defaults to `TRUE`.

Conditions are evaluated

1. when the state is entered (after entry block execution), and
2. when an event happens (see below).

Evaluation proceeds in the order in which the `transitions` appear in the program. If one of the `conditions` evaluates to `true`, the corresponding action block is executed, any exit block of the state is executed, and the state changes to the specified new state. Otherwise, the state set waits until an event happens.

There are five types of event:

- a process variable monitor is posted
- an asynchronous `pvGet` or `pvPut` completes
- a `delay` timer expires
- an event flag is set or cleared
- a process variable connects or disconnects

While the sequencer's runtime system and the compiler work together to minimize evaluation of conditions, there is no guarantee whatsoever about the number of times conditions might be evaluated before one of them returns `true`.

Conditions are usually written so that they have no side-effects. This ensures that it does not matter how often they are evaluated or in which order.

Note: If a condition has a side-effect, care should be taken to ensure that the effect is limited to the case when the whole(!) condition fires (returns `true`), so that it will occur at most once. Otherwise you will have no control over when and how often the effect happens.

Built-in functions that have side-effects and are suitable for use in conditions, notably `efTestAndClear` and `pvGetQ`, meet this criterion, but only of used by themselves alone. If used as part of a larger expression, the whole condition might no longer meet the citerion.

New in version 2.1.

```
transition  ::=   "when" "(" condition ")" block "exit"
```

Instead of declaring which should be the next state, one can use the single keyword "exit". This has the same effect as a call to the `seqStop` shell command, that is, at this point all state sets should terminate (after completing any action block in progress) and execution proceed with the global exit block (if any). Afterwards all channels are disconnected, all allocated memory is freed, and the program terminates.

Note: If the program has been started under an ioc shell, then only the SNL program is terminated, not the whole ioc. If terminating the ioc shell is required, you should call the `exit()` function from the standard C library. This call can conveniently be placed in the SNL program's global exit block.

7.4.3 Block

```
block        ::=   "{" block_defns statements "}"
block_defns  ::=   block_defns block_defn
block_defns  ::=
block_defn   ::=   declaration
```

```
block_defn   ::=   c_code
```

Blocks are enclosed in matching (curly) braces. They may contain any number of block definitions and afterwards any number of statements.

Block definitions are: declarations and embedded C code.

7.5 Statements and Expressions

7.5.1 Statements

```
statements      ::=   statements statement
statements      ::=
statement       ::=   "break" ";"
statement       ::=   "continue" ";"
statement       ::=   c_code
statement       ::=   block
statement       ::=   "if" "(" comma_expr ")" statement
statement       ::=   "if" "(" comma_expr ")" statement "else" statement
statement       ::=   "while" "(" comma_expr ")" statement
statement       ::=   for_statement
statement       ::=   opt_expr ";"
for_statement   ::=   "for" "(" opt_expr ";" opt_expr ";" opt_expr ")" statement
```

As can be seen, most C statements are supported. Not supported is the switch/case statement.

New in version 2.1.

```
statement   ::=   "state" identifier ";"
```

The *state change statement* is not borrowed from C; it is only available in the action block of a state `transition` and has the effect of immediately jumping out of the action block, overriding the statically specified new state (given after the block) with its state argument.

New in version 2.2.

```
statement   ::=   "return" opt_expr ";"
```

The return statement can be used only inside Function Definitions.

7.5.2 Expressions

Formation rules for expressions are listed in groups of descending order of precedence. Precedence is determined by the standard C operator precedence table.

Primary Expressions

```
expr    ::=   integer_literal
expr    ::=   floating_point_literal
expr    ::=   string
expr    ::=   variable
```

```
string  ::=   string_literal
expr    ::=   "(" comma_expr ")"
```

These are literals, variables, and parenthesized expressions.

Postfix Operators

```
expr    ::=   expr "(" args ")"
expr    ::=   "exit" "(" args ")"
expr    ::=   expr "[" expr "]"
expr    ::=   expr "."  member
expr    ::=   expr "->" member
expr    ::=   expr "++"
expr    ::=   expr "-"
member  ::=   identifier
```

These are all left associative.

Note: `exit` must be listed explicitly because in SNL it is a keyword, not an identifier. The extra rule allows calls to the standard library function `exit`.

Prefix Operators

```
expr  ::=   "+" expr
expr  ::=   "-" expr
expr  ::=   "*" expr
expr  ::=   "&" expr
expr  ::=   "!"  expr
expr  ::=   "~" expr
expr  ::=   "++" expr
expr  ::=   "-" expr
```

Prefix operators are right-associative.

New in version 2.2: Type casts and sizeof operator:

```
expr  ::=   "(" type_expr ")" expr
expr  ::=   "sizeof" "(" type_expr ")"
expr  ::=   "sizeof" expr
```

Binary Operators

```
expr  ::=   expr "-" expr
expr  ::=   expr "+" expr
```

```
expr  ::=   expr "<<" expr
expr  ::=   expr ">>" expr
```

```
expr  ::=   expr "*" expr
expr  ::=   expr "/" expr
```

```
expr  ::=   expr ">" expr
expr  ::=   expr ">=" expr
expr  ::=   expr "<=" expr
expr  ::=   expr "<" expr
```

```
expr  ::=   expr "==" expr
expr  ::=   expr "!=" expr
```

```
expr  ::=   expr "&" expr
```

```
expr  ::=   expr "^" expr
```

```
expr  ::=   expr "|" expr
```

```
expr  ::=   expr "||" expr
```

```
expr  ::=   expr "&&" expr
```

```
expr  ::=   expr "%" expr
```

All binary operators associate to the left.

Note: Like in most programming languages, including C, evaluation of conditional expressions using `&&` and `||` is done *lazily*: the second operand is not evaluated if evaluation of the first already determines the result. This particularly applies to the boolean expressions in `transition` clauses. See the built-in function `pvGetQ` for an extended discussion.

Ternary Operator

```
expr  ::=   expr "?"  expr ":"  expr
```

The ternary operator (there is only one) is right-associative.

Assignment Operators

```
expr  ::=   expr "=" expr
expr  ::=   expr "+=" expr
```

```
expr  ::=   expr "-=" expr
expr  ::=   expr "&=" expr
expr  ::=   expr "|=" expr
expr  ::=   expr "/=" expr
expr  ::=   expr "*=" expr
expr  ::=   expr "%=" expr
expr  ::=   expr "<<=" expr
expr  ::=   expr ">>=" expr
expr  ::=   expr "^=" expr
```

These operators are right-associative.

Comma Operator

```
comma_expr  ::=   comma_expr "," expr
comma_expr  ::=   expr
opt_expr    ::=   comma_expr
opt_expr    ::=
```

The comma operator is left associative. An `opt_expr` is an optional `comma_expr`; it appears, for instance, inside a `for_statement`.

Argument List

```
args  ::=   args "," expr
args  ::=   expr
args  ::=
```

Function argument lists look exactly like chained application of the comma operator, which is why application of the comma operator in an argument list must be grouped by parentheses.

7.6 Built-in Constants

Some of the built-in functions use (i.e. accept or return) values of certain enumeration types and constants. These are also available to C code and defined in the header files pvAlarm.h and seqCom.h.

Changed in version 2.1.6.

The pvStat and pvSevr constants are now known to the compiler, so you no longer have to declare them as `foreign` in the SNL code.

pvStat

An enumeration for status values.

```
typedef enum {
    /* generic OK and error statuses */
    pvStatOK            = 0,
    pvStatERROR         = -1,
    pvStatDISCONN       = -2,

    /* correspond to EPICS statuses */
    pvStatREAD          = 1,
    pvStatWRITE         = 2,
    pvStatHIHI          = 3,
```

```
    pvStatHIGH         = 4,
    pvStatLOLO         = 5,
    pvStatLOW          = 6,
    pvStatSTATE        = 7,
    pvStatCOS          = 8,
    pvStatCOMM         = 9,
    pvStatTIMEOUT      = 10,
    pvStatHW_LIMIT     = 11,
    pvStatCALC         = 12,
    pvStatSCAN         = 13,
    pvStatLINK         = 14,
    pvStatSOFT         = 15,
    pvStatBAD_SUB      = 16,
    pvStatUDF          = 17,
    pvStatDISABLE      = 18,
    pvStatSIMM         = 19,
    pvStatREAD_ACCESS  = 20,
    pvStatWRITE_ACCESS = 21
} pvStat;
```

pvSevr

```
typedef enum {
    /* generic OK and error severities */
    pvSevrOK      = 0,
    pvSevrERROR   = -1,

    /* correspond to EPICS severities */
    pvSevrNONE    = 0,
    pvSevrMINOR   = 1,
    pvSevrMAJOR   = 2,
    pvSevrINVALID = 3
} pvSevr;
```

compType

```
enum compType {
    DEFAULT,
    ASYNC,
    SYNC
};
```

Note: Only `SYNC` and `ASYNC` are SNL built-in constants (i.e. known to `snc`). The constant `DEFAULT` is for use in C code (to represent a missing optional argument).

---

seqBool

```
typedef int seqBool;

#define TRUE   1
#define FALSE  0
```

NOEVFLAG

```
#define NOEVFLAG 0
```

This can be given as second argument to `pvSync` (instead of an event flag) to cancel the sync.

7.7 Built-in Functions

The following special functions are built into the language. In most cases the compiler performs some special interpretation of the arguments to these functions. Using their C equivalents in escaped C code is possible but subject to special rules.

For documentation purposes only, we assign special pseudo-types to some of the parameters of built-in functions. Note that these are not valid SNL types.

`channel`

A parameter with this type means the function expects an SNL variable or array element that is assigned to a single (possibly anonymous) process variable.

A parameter with type `channel[]` means the function expects a channel array, that is, an array where the elements are assigned to separate (possibly anonymous) process variables. By convention, functions accepting a `channel` array have a name that starts with "pvArray".

Several of these functions are intended to be called only from `transition` clauses or only from action code. If the compiler allows it, it is safe to call them in another context, but the effect might not be what you expect.

7.7.1 delay

seqBool **`delay`** (double *delay_in_seconds*)

Returns whether the specified time has elapsed since entering the state. It should be used only within a `transition` expression.

The "t" *State Option* controls whether the delay is measured from when the current state was entered from a different state (*-t*) or from any state, including itself (*+t*, the default).

Changed in version 2.2.

It is no longer allowed to call this function outside the `condition` of a `transition` clause.

7.7.2 pvPut

pvStat **`pvPut`** (channel *ch*, compType mode = *DEFAULT*, double timeout = *10.0*)

Puts (or writes) the value of `ch` to the process variable it has been assigned to. Returns the status from the PV layer (e.g. `pvStatOK` for success).

By default, i.e. with no optional arguments, `pvPut` is un-confirmed "fire and forget"; completion must be inferred by other means. An optional second argument can change this default:

- `SYNC` causes it to block the state set until completion. This mode is called *synchronous*.
- `ASYNC` allows the state set to continue but still check for completion via a subsequent call to `pvPutComplete` (typically in a `condition`). This mode is called *asnchronous*.

A timeout value may be specified after the `SYNC` argument. This should be a positive floating point number, specifying the number of seconds before the request times out. This value overrides the default timeout of 10 seconds.

Note that SNL allows only one pending `pvPut` per variable and state set to be active. As long as a `pvPut(var,ASYNC)` is pending completion, further calls to `pvPut(var,ASYNC)` from the same state set immediately fail and an error message is printed; whereas further calls to `pvPut(var,SYNC)` are *delayed* until the previous operation completes. Thus

```
pvPut(var,ASYNC);
pvPut(var,SYNC);
```

and

```
pvPut(var,SYNC);
pvPut(var,SYNC);
```

are equivalent. Whereas in

```
pvPut(var,ASYNC);
pvPut(var,ASYNC);
```

the second pvPut may fail if it is a named PV located on another IOC (since the first one is still awaiting completion), whereas for local named PVs and anonymous PVs it will succeed (since completion is immediate in these cases).

In *Safe Mode*, `pvPut` can be used with anonymous PVs (variables assigned to "") to communicate between state sets. This makes sense only with global variables as only those can be referenced in more than one state set. The behaviour for anonymous PVs exactly mirrors that of named PVs, including the fact that the new value will not be seen by other state sets until they issue a pvGet, or, if the variable is monitored, until they wait for an event in a `condition`. Note that for anonymous PVs completion is always immediate, so the `ASYNC` option is not very useful.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.3 pvPutComplete

seqBool **pvPutComplete** (channel *ch*)

Returns whether the last asynchronous `pvPut` to this process variable has completed.

Always returns `true` for anonymous PVs.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error. In previous versions this was the one built-in pv function that correctly worked on channel arrays. The corresponding C function still has the extra arguments for compatibility, but in SNL code you must now use `pvArrayPutComplete` for channel arrays.

7.7.4 pvArrayPutComplete

New in version 2.2.

seqBool **pvArrayPutComplete** (channel *ch[]*, unsigned int *length*, seqBool any = *FALSE*, seqBool *complete = *NULL*)

Like `pvPutComplete`, but returns whether all of the `length` first elements of the array have completed their last asynchronous `pvPut`. The `length` argument should not be larger than the number of elements in the array.

If the optional `any` argument is `true`, then instead it returns whether *any* of these channels have completed.

If the optional `complete` argument is not `NULL`, it must point to an array of `seqBool`s with at least `length` elements; the will then write the individual results for the array elements into this array.

7.7.5 pvPutCancel

New in version 2.2.

void **pvPutCancel** (channel *ch*)

Cancel a pending (asynchronous) `pvPut`.

7.7.6 pvArrayPutCancel

New in version 2.2.

void **pvArrayPutCancel** (channel *ch[]*, unsigned int *length*)

Like `pvPutCancel` but all pending asynchronous `pvPut`s to the first `length` elements of the array are cancelled.

7.7.7 pvGet

pvStat **pvGet** (channel *ch*, compType ct = *DEFAULT*, double timeout = *10.0*)

Gets (or reads) the value of `ch` from the process variable it has been assigned to. Returns the status from the PV layer (e.g. `pvStatOK` for success).

With no optional arguments, the completion type is determined by the value of a program option: if `-a` is in effect (the default), then the state set will block until the read operation is complete or until the default timeout of 10 seconds has expired; if `+a` is in effect, the the call completes immediately and completion can be checked with subsequent calls to `pvGetComplete` (typically in a `condition`).

An optional second argument overrides the program option: `SYNC` makes the call block and `ASYNC` makes the call return immediately, regardless of whether `-a` or `+a` is in effect.

In *Safe Mode*, if `ASYNC` is specified and the variable is not monitored, then the state set local copy of the variable will not be updated until a call to pvGetComplete is made and returns `true` or, if the variable is monitored, until the state set waits for events in a `transition` clause. Note that anonymous PVs behave exactly in the same way.

Like for `pvPut`, only one pending `pvGet` per channel and state set can be active.

Changed in version 2.2.

A timeout value may be specified after the `SYNC` argument. This should be a positive floating point number, specifying the number of seconds before the request times out. This value overrides the default timeout of 10 seconds.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.8 pvGetComplete

seqBool **pvGetComplete** (channel *ch*)

Returns whether the last asynchronous `pvGet` for `ch` has completed.

In *Safe Mode*, the the state set local copy of the variable will be updated with the value from the PV layer as a side effect of this call (if `true` is returned).

Always returns `true` for anonymous PVs.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error. For use with multi-PV arrays there is now `pvArrayGetComplete`.

7.7.9 pvArrayGetComplete

New in version 2.2.

seqBool **pvArrayGetComplete** (channel *ch[]*, unsigned int *length*, seqBool any = *FALSE*, seqBool *complete = *NULL*)

Like pvGetComplete but works on all channels of a multi-PV array. The optional arguments have the same meaning as for pvArrayPutComplete.

7.7.10 pvGetCancel

New in version 2.2.

void **pvGetCancel** (channel *ch*)

Cancel a pending (asynchronous) pvGet.

7.7.11 pvArrayGetCancel

New in version 2.2.

void **pvArrayGetCancel** (channel *ch[]*, unsigned int *length*)

Cancel pending (asynchronous) pvGet operations for the first length elements of the array.

7.7.12 pvGetQ

seqBool **pvGetQ** (channel *ch*)

The channel argument must have been associated with a queue using the syncq clause.

If the queue is not empty, remove its first (oldest) value and update the variable with it. Returns whether there was an element in the queue (and the variable got updated). If an element gets removed, and the queue becomes empty as a result, then any event flag synced to the variable will be cleared.

Note that since pvGetQ may have a *side-effect* you should be careful when combining a call to pvGetQ with other conditions in the same transition clause, e.g.

```
when (pvGetQ(msg) && other_condition) {
   printf(msg);
} state ...
```

would remove the head from the queue every time the condition gets evaluated, regardless of whether other_condition is true. This is most probably not the desired effect, as you would lose an unknown number of messages. (Of course it *could* be exactly what you want, for instance if other_condition were something like !suppress_output.) Whereas

```
when (other_condition && pvGetQ(msg)) {
   printf(msg);
} state ...
```

is "safe", in the sense that no messages will be lost. BTW, If you combine with a disjunction ("||") it is the other way around, i.e. pvGetQ should appear as the first operand. This is all merely a result of the evaluation order imposed by the C language (and thus by SNL); similar remarks apply whenever you want to use an expression inside a transition clause that potentially has a side-effect.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.13 pvFreeQ

void **pvFreeQ** (queued_channel)

Deletes all entries from a queued variable's queue and clears the associated event flag.

Changed in version 2.1.

Queue elements are no longer dynamically allocated, so this is now an alias for pvFlushQ.

7.7.14 pvFlushQ

void **pvFlushQ**(queued_channel)

New in version 2.1.

Flush the queue associated with this variable, so it is empty afterwards.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.15 pvAssign

pvStat **pvAssign**(channel *ch*, char **pv_name*)

Assigns or re-assigns `ch` to a process variable with name `pv_name`. If `pv_name` is an empty string or `NULL`, then `ch` is de-assigned (not associated with any process variable); in *Safe Mode* it causes assignment to an anonymous PV.

See also `assign` clause.

Note that pvAsssign is *asynchronous*: it sends a request to search for and connect to the given `pv_name`, but it does not wait for a response, similar to `pvGet(var,ASYNC)`. Calling pvAssign *does* have one immediate effect, namely de-assigning the variable from any PV it currently is assigned to. In order to make sure that it has connected to the new PV, you can use the `pvConnected` built-in function inside a `transition` clause.

Like for all other operations, completion is immediate for anonymous PVs.

Note: pvAssign can only be called on variables (or array elements) that have been statically marked as process variables using the `assign` syntax. An empty string may be used for the initial assignment, or (from version 2.1. onward) the simplified form `assign var`.

Warning: If a variable gets de-assigned from a non-empty to an empty name, the corresponding channel is destroyed, which means that dynamically allocated memory gets freed. If your system cannot handle dynamic memory allocation without fragmentation, care should be taken that assignment and de-assignment do not alternate too often.

Note: If you want to assign array elements to separate PVs, you cannot currently do this with a single call (in contrast to doing it in an `assign` clause. Instead, you must call `pvAssign` for each array element individually.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.16 pvAssignSubst

pvStat **pvAssignSubst**(channel *ch*, char **pv_name*)

Like `pvAssign`, except that it substitutes program parameters, like the `assign` clause.

7.7.17 pvMonitor

pvStat **pvMonitor**(channel *ch*)

Initiates a monitor on the process variable that `ch` was assigned to.

See `monitor` clause.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.18 pvArrayMonitor

New in version 2.2.

pvStat **pvArrayMonitor** (channel *ch[]*, unsigned int *length*)

Like `pvMonitor` but turns on monitors for the first `length` elements of a channel array.

See `monitor` clause.

7.7.19 pvStopMonitor

pvStat **pvStopMonitor** (channel *ch*)

Terminates a monitor on the underlying process variable.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.20 pvArrayStopMonitor

New in version 2.2.

pvStat **pvArrayStopMonitor** (channel *ch[]*, unsigned int *length*)

Like `pvStopMonitor` but turns off monitors for the first `length` elements of a channel array.

7.7.21 pvSync

void **pvSync** (channel *ch*, evflag *ef*)

Synchronizes a variable with an event flag, or removes such a synchronization if `ef` is `NOEVFLAG`.

See `sync` clause.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.22 pvArraySync

New in version 2.2.

void **pvArraySync** (channel *ch[]*, unsigned int *length*, evflag *ef*)

Synchronizes the first `length` elements of a channel array with an event flag, or removes such a synchronization if `ef` is `NOEVFLAG`.

See `sync` clause.

7.7.23 pvCount

unsigned **pvCount** (channel *ch*)

Returns the element count associated with the (single) process variable. This value is independent of the array size given in the declaration, it can be smaller or larger.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.24 pvStatus

pvStat **pvStatus** (channel *ch*)

Returns the current alarm status of the underlying PV or indicates failure of a previous PV operation.

The status, severity, and message returned by `pvStatus`, `pvSeverity`, and `pvMessage` reflect either the underlying PV's properties (if a `pvPut` or `pvGet` operation completed, or the variable is `monitor`ed), or else indicate a failure to initiate one of these operations.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.25 pvSeverity

pvSevr **pvSeverity** (channel *ch*)

Returns the current alarm severity (e.g. pvSevrMAJOR) of the underlying PV or indicates failure of a previous PV operation.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.26 pvMessage

const char ***pvMessage** (channel *ch*)

Returns the current error message of the variable, or "" (the empty string) if none is available.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.27 pvTimeStamp

struct epicsTimeStamp **pvTimeStamp** (channel *ch*)

Returns the time stamp for the last `pvGet` completion or monitor event for this variable.

New in version 2.2.

You can declare variables of type `struct epicsTimeStamp` directly in SNL code.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.28 pvAssigned

seqBool **pvAssigned** (channel *ch*)

Returns whether the SNL variable is currently assigned to a process variable. Note that this function returns `FALSE` for anonymous PVs.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.29 pvConnected

seqBool **pvConnected**(channel *ch*)

Returns whether the underlying process variable is currently connected.

Changed in version 2.2.

Always returns `true` for anonymous PVs.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.30 pvArrayConnected

New in version 2.2.

seqBool **pvArrayConnected**(channel *ch[]*, unsigned int *length*)

Returns whether elements of a channel array are currently connected.

7.7.31 pvIndex

unsigned **pvIndex**(channel *ch*)

Returns the index associated with a variable. See *Calling PV Functions from C* for how to use this function.

Changed in version 2.2.

Calling this function with a multi-PV array is no longer allowed and results in a compile-time error.

7.7.32 pvFlush

void **pvFlush**()

Causes the PV layer to flush its send buffer. This is only necessary if you need to make sure that CA operations are started *before* the action block finishes. The buffer is always automatically flushed before the sequencer waits for events, that is, after a state's entry block is executed (or on entry to the state if there is no entry block). The buffer is also flushed after initiating a synchronous operation that waits for a callback (i.e. `pvPut(var,SYNC)` and `pvGet(var,SYNC)`).

7.7.33 pvChannelCount

unsigned **pvChannelCount**()

Returns the total number of process variables associated with the program.

7.7.34 pvAssignCount

unsigned **pvAssignCount**()

Returns the total number of SNL variables in this program that are assigned to underlying process variables.

For instance, if all SNL variables are assigned then the following expression is `true`:

```
pvAssignCount() == pvChannelCount()
```

Each element of an SNL array counts as variable for the purposes of `pvAssignCount`.

7.7.35 pvConnectCount

unsigned **`pvConnectCount`**`()`

Returns the total number of underlying process variables that are connected.

For instance, if all assigned variables are connected then the following expression is `true`:

```
pvConnectCount() == pvAssignCount()
```

7.7.36 efSet

void **`efSet`**(evflag *ef*)

Sets the event flag and causes evaluation of the `transition` clauses for all state sets that are pending on this event flag.

7.7.37 efClear

seqBool **`efClear`**(evflag *ef*)

Clears the event flag and causes evaluation of the `transition` clauses for all state sets that are pending on this event flag.

7.7.38 efTest

seqBool **`efTest`**(evflag *ef*)

Returns whether the event flag was set.

Note: In safe mode, this function is a synchronization point for all variables (channels) that are `sync`ed with it, i.e. the state set local copy of these variables will be updated from their current globally visible value.

7.7.39 efTestAndClear

seqBool **`efTestAndClear`**(evflag *ef*)

Clears the event flag and returns whether the event flag was set. It is intended for use within a `transition` clause.

Note: In safe mode, this function is a synchronization point for all variables (channels) that are `sync`ed with it, i.e. the state set local copy of these variables will be updated from their current globally visible value.

7.7.40 macValueGet

char* **`macValueGet`**(char **parameter_name*)

Returns a pointer to the value of the specified program parameter, if it exists, else `NULL`. See Program Name and Parameters.

7.7.41 optGet

seqBool **optGet** (char **option_name*)

Returns whether the program option with the given name is in effect or not. The option name is a string consisting of a single letter such as "a" for the program option `+a`/`-a`.

7.8 Safe Mode

New in version 2.1.

SNL code can be interpreted in safe mode by using the program option `+s`, see *Safe Mode* for a less dense introduction. Safe mode implies reentrant mode, see *Variable Modification for Reentrant Option*.

In safe mode, global variables are treated as if they were local to the state set. Changing such a variable inside an action block does not have any immediate effect on other state sets, nor do external events, such as monitors or pvGet completions, change the variables as seen by a state set, except at well defined *Synchronization Points*. In other words, each state set works on its own copy ("view") of all variables. The `Channel Access` layer (and the part of the run time system that implements *Anonymous Channels*) acts on yet another copy, the "world view". Information is only ever exchanged between the world view and the state set local view. All communication between state sets must be explicit: apart from using event flags, the only way information can flow *out* of a state set is using `pvPut`, the only way *in* is via `pvGet` or monitors.

An important (and welcome) side-effect of safe mode is that in the action block after a condition you can rely on the `condition` to be `true` (and remain `true` – unless, of course, the action block itself invalidates it by modifying the variables involved).

7.8.1 Synchronization Points

At certain points in the program a state set's view of the variables is updated from the world view. These are:

- in each state, immediately before `condition`s are evaluated (for monitored channels), and
- in calls to some of the built-in functions: `efTest`, `efTestAndClear`, `pvGet` (in SYNC mode), and `pvGetComplete` (when it signals completion).

Note that there is no point at which variables modified by a state set are automatically "published". For this you have to use `pvPut` explicitly, which updates the world view as a side-effect.

7.8.2 Anonymous Channels

It would be impractical if the programmer had to create a real PV for each communication channel inside a program. Instead, in safe mode you can use an *anonymous channel* for this purpose. Anonymous channels are created with the usual `assign` clause, but use an empty string (`""`) for the PV name (which in traditional mode is interpreted as "not assigned"). You can imagine such channels to be connected to "virtual" PVs that are implemented inside the sequencer's run time system.

Anonymous channels should work exactly like channels assigned to a "real" PV – they can be `monitor`ed, `sync`ed to event flags etc. – except that they are always connected and the operations on them always complete immediately.

7.8.3 Event Flags in Safe Mode

Apart from the `sync` and `syncq` features, and apart from the atomicity of `efTestAndClear`, an event flag behaves like an anonymous PV of boolean type with a monitor. In safe mode, `efTest` and `efTestAndClear` have the additional side-effect of acting as a synchronization point for all variables that are `sync`ed with the event flag.

7.9 Foreign Variables and Functions

You can use (reference, call) any exported C variable or function that is linked to your program. For instance, you may link a C library to the SNL program and simply call functions from it directly in SNL code. We call such entities or objects "foreign".

It is advisable to take care that the C code generated from your SNL program contains (directly or via `#include`) a valid declaration for foreign entities, otherwise the C compiler will not be able to catch type errors (the SNL compiler does not do any type-checking by itself). For libraries, this is usually done by adding a line

```
%%#include "api-of-your-library.h"
```

(See also *Preprocessor Directives*.)

Note that when passing channel variables (i.e. ones assigned to one or more PVs using an `assign` clause) to a C function, only the *value* is passed. The special assigned" status of the variable gets lost as soon as the function is entered (same as for functions defined in SNL).

Changed in version 2.2.

The SNL compiler no longer complains about "undeclared identifiers" when using foreign variables unless you supply the `+W` (extra warnings) option.

CHAPTER

EIGHT

EXAMPLES OF SNL PROGRAMS

8.1 Entry and exit action example

The following program illustrates entry and exit actions.

```
program snctest
float v;
assign v to "grw:xxxExample"; monitor v;

ss ss1 {
  state low {
    entry {
      printf("Will do this on entry");
      printf("Another thing to do on entry");
    }
    when (v>5.0) {
      printf("now changing to high\n");
    } state high
    when (delay(.1)) { } state low
    exit {
      printf("Something to do on exit");
    }
  }

  state high {
    when (v<=5.0) {
      printf("changing to low\n");
    } state low
    when(delay(.1)) { } state high
  }
}
```

8.2 Dynamic assignment example

The following segment of a program illustrates dynamic assignment of database variables to database channels. We have left out error checking for simplicity.

```
program dynamic
option -c; /* don't wait for db connections */
string sysName;
assign sysName to "";

long setpoint[5];
assign setpoint to {}; /* don't need all five strings */

int i;
char str[30];
```

```
ss dyn {
  state init {
    when () {
      sprintf (str, "MySys:%s", "name");
      pvAssign (sysName, str);
      for (i = 0; i < 5; i++) {
        sprintf (str, "MySys:SP%d\n", i);
        pvAssign (setpoint[i], str);
        pvMonitor (setpoint[i]);
      }
    } state process
  }

  state process {
    ...
  }
}
```

8.3 Complex example

The following program contains most of the concepts presented in the previous sections. It consists of four state sets: (1) `level_det` , (2) `generate_voltage` , (3) `test_status` , and (4) `periodic_read` . The state set `level_det` is similar to the example in *A Complete Program* from the *Tutorial*. It generates a triangle waveform in one state set and detects the level in another. Other state sets detect and print alarm status and demonstrate asynchronous `pvGet` and `pvPut` operation. The program demonstrates several other concepts, including access to run-time parameters with macro substitution and `macValueGet` , use of arrays, escaped C code, and VxWorks input-output.

8.3.1 Preamble

```
program example ("unit=ajk, stack=11000")

/*================== declarations ========================*/
float ao1;
assign ao1 to "{unit}:ao1";
monitor ao1;

float ao2;
assign ao2 to "{unit}:ao1";

float wf1[2000];
assign wf1 to "{unit}:wf1.FVAL";

short bi1;
assign bi1 to "{unit}:bi1";

float delta;
short prev_status;
short ch_status;

evflag ef1;
evflag ef2;

option +r;

int fd; /* file descriptor for logging */
char *pmac; /* used to access program macros */
```

8.3.2 level_det state set

```
/*=================== State Sets ============================*/

/* State set level_det detects level > 5v & < 3v */
ss level_det {

  state start {
    when() {
      fd = -1;
      /* Use parameter to define logging file */
      pmac = macValueGet("output");
      if (pmac == 0 || pmac[0] == 0) {
        printf("No macro defined for \"output\"\n");
        fd = 1;
      }
      else {
        fd = open(pmac, (O_CREAT | O_WRONLY), 0664);
        if (fd == ERROR) {
          printf("Can't open %s\n", pmac);
          exit (-1);
        }
      }
      fdprintf(fd, "Starting program\n");
    } state init
  }

  state init {
    /* Initialize */
    when (pvConnectCount() == pvChannelCount() ) {
      fdprintf(fd, "All channels connectedly");
      bi1 = FALSE;
      ao2 = -1.0;
      pvPut(bi1);
      pvPut(ao2);
      efClear(ef2);
      efSet(ef1);
    } state low

    when (delay(5.0)) {
      fdprintf(fd, "...waiting\n");
    } state init
  }

  state low {
    when (ao1 > 5.0) {
      fdprintf(fd, "High\n");
      bi1 = TRUE;
      pvPut(bi1);
    } state high

    when (pvConnectCount() < pvChannelCount() ) {
      fdprintf(fd, "Connection lost\n");
      efClear(ef1);
      efSet(ef2);
    } state init
  }

  state high {
    when (ao1 < 3.0) {
      fdprintf(fd, "Low\n");
      bi1 = FALSE;
      pvPut(bi1);
```

```
    } state low

    when (pvConnectCount() < pvChannelCount() ) {
      efSet(ef2);
    } state init
  }
}
```

8.3.3 generate_voltage state set

```
/* Generate a ramp up/down */
ss generate_voltage {
  state init {
    when (efTestAndClear(ef1)) {
      printf("start ramp\n");
      fdprintf(fd, "start ramp\n");
      delta = 0.2;
    } state ramp
  }

  state ramp {
    when (delay(0.1)) {
      if ( (delta > 0.0 && ao2 >= 11.0) ||
        (delta < 0.0 && ao2 <= -11.0) ) {
        delta = -delta;
      }
      ao2 += delta;
      pvPut(ao2);
    } state ramp

    when (efTestAndClear(ef2)) {
    } state init
  }
}
```

8.3.4 test_status state set

```
/* Check for channel status; print exceptions */
ss test_status {
  state init {
    when (efTestAndClear(ef1)) {
      printf("start test_status\n");
      fdprintf(fd, "start test_status\n");
      prev_status = pvStatus(ao1);
    } state status_check
  }

  state status_check {
    when ((ch_status = pvStatus(ao1)) != prev_status) {
      print_status(fd, ao1, ch_status, pvSeverity(ao1));
      prev_status = ch_status;
    } state status_check
  }
}
```

8.3.5 periodic_read state set

```
/* Periodically write/read a waveform channel. This uses
   pvGetComplete() to allow asynchronous pvGet().
*/
ss periodic_read {
  state init {
    when (efTestAndClear(ef1)) {
      wf1[0] = 2.5;
      wf1[1] = -2.5;
      pvPut(wf1);
    } state read_chan
  }

  state read_chan {
    when (delay(5.)) {
      wf1[0] += 2.5;
      wf1[1] += -2.5;
      pvPut(wf1);
      pvGet(wf1);
    } state wait_read
  }

  state wait_read {
    when (pvGetComplete(wf1)) {
      fdprintf(fd, "periodic read: ");
      print_status(fd, wf1[0], pvStatus(wf1), pvSeverity(wf1));
    } state read_chan
  }
}
```

8.3.6 exit procedure

```
/* Exit procedure - close the log file */
exit {
  printf("close fd=%d\n", fd);
  if ((fd > 0) && (fd != ioGlobalStdGet(1)) )
    close(fd);
  fd = -1;
}
```

8.3.7 C functions

```
/*==================== End of state sets ====================*/

%{
  /* This C function prints out the status, severity,
  and value for a channel. Note: fd is passed as a
  parameter to allow reentrant code to be generated */
  print_status(int fd, float value, int status, int severity)
  {
    char *pstr;

    switch (status)
    {
      case NO_ALARM:    pstr = "no alarm";        break;
      case HIHI_ALARM:  pstr = "high-high alarm"; break;
      case HIGH_ALARM:  pstr = "high alarm";      break;
      case LOLO_ALARM:  pstr = "low-low alarm";   break;
```

```
      case LOW_ALARM:   pstr = "low alarm";       break;
      case STATE_ALARM: pstr = "state alarm";     break;
      case COS_ALARM:   pstr = "cos alarm";       break;
      case READ_ALARM:  pstr = "read alarm";      break;
      case WRITE_ALARM: pstr = "write alarm";     break;
      default:          pstr = "other alarm";     break;
    }
    fprintf (fd, "Alarm condition: \"%s\"", pstr);
    if (severity == MINOR_ALARM)
      pstr = "minor";
    else if (severity == MAJOR_ALARM)
      pstr = "major";
    else
      pstr = "none";
    fdprintf (fd, ", severity: \"%s\", value=%g\n", pstr, value);
  }
}%
```

CHAPTER

NINE

RELEASE NOTES FOR VERSION 2.2

9.1 Release 2.2.5

- docs: use RMDIR instead of rm -rf for clean target in the Makefile

  This fixes (harmless but annoying) error messages on Windows.

- snc: use mkmf.pl to generate the dependencies

  This fixes error messages and sometimes also build failures during parallel builds on Linux machines iwth many cores. These are caused by EPICS build rules not passing -MD to gcc when generating dependencies for C source files.

- snc: fixed C90 incompatibility in the parser template, see *Known Problems in Release 2.2.4*.

9.2 Release 2.2.4

code:

- seq: fixed a possible race condition with pvAssign

  The problem concerns access to the dbch member of CHAN: the runtime pvAssign may free this pointer (and also sets it to NULL); on the other hand, the CA callbacks need to access it, or else must return immediately. There was a test that checks if dbch is NULL but it was not locked against concurrent access from pvAssign, which could lead to an assertion failure or worse to invalid memory accesses.

- seq: in pvFlushQ test that queue and associated event flag of the channel actually exist before accessing them.

  This fixes possible crashes when using this function with non-queued or queued but non-synced channels.

- pv: fix test for result of epicsTimeGetCurrent

  This change fixes a problem with EPICS base-3.16, which adds more error codes to epicsTime.

- lemon: updated lemon.c and the parser template from upstream

  This was done to avoid 64 bit problems that the native Windows compiler warned about, since such problems are usually handled quite efficiently by the sqlite maintainers. Unfortunately the latest version of the parser template had a serious bug that could result in compiler crashes. This bug has been fixed in the sequencer (and reported, but not yet fixed, in the upstream version). This in turn uncovered a weakness in the compiler tests, see below.

docs:

- fixed errors in section “Asynchronous Use of pvPut” of tutorial

  Thanks to Christian Pulvermacher <christian.pulvermacher@kek.jp> for spotting this.

build system:

- snc: turn LEMON into an absolute path with base 3.15

  This fixes a build problem on some Windows versions which don't like relative path names for commands when the output is redirected to a file.

- snc: extra dependency avoids (harmless) errors when generating .d files
- snc: simplify the multi-target lemon rule

  Instead of generating an intermediate file ("parser_created") we now use a pattern rule. This is the more reliable (and recommended) way to make rules with multiple targets work as expected.

- test only for BASE_3_14==YES

  Testing for 3.15 doesn't make sense (yet), as the majority of changes will be needed for 3.16 and later versions, too. Testing BASE_3_15==YES will be reserved for incompatibilities between 3.15 and 3.16 etc.

- avoid deprecation warnings for use of PATH_FILTER with base 3.15
- snc: make recipe for snl.bnf atomic

test/examples:

- moved test for invalid state change statement to compiler tests
- moved demo programs from test/validate to examples/small

  The programs are only compiled as TESTPROD_HOST to avoid cluttering the install directories any further; db files can be served using the standard softIoc binary from base.

- added timer.st to examples
- refactor killing background IOCs
- test that snc terminates normally on all input programs

  Previously, in case snc crashes on some input file, it was possible that this was not detected by the tests.

9.3 Release 2.2.3

Thanks to the new co-maintainer Freddie Akeroyd <freddie.akeroyd@stfc.ac.uk> this release fixes building and running the tests for Windows7 in 32 and 64 bit, both with cygwin and native compiler. Please note that parallel building may not work on Windows yet.

A few bugs have been fixed and some minor improvements made here or there, for details see the list below.

snc:

- cast 0 to Node* in varargs call to node constructor
- added a missing include to types.h
- explain apparently missing var init in a comment
- fix warnings and error when DEBUG is defined

seq:

- Initialise timeNow

  Required for running tests with windows debug build

- fix default put during a pending async put

  Attempting to issue a DEFAULT pvPut even when an ASYNC put is already pending on the same variable led to an assertion failure. This is now fixed.

common:

- cleaned up the prim_types mess, snc no longer depends on seq library

  Making the compiler depend on the seq library was a bad idea, since the names of the primitive types were the only thing that snc needed from seq.

  Instead I added a new src/common subdir that installs (and for seq_release.h generates) the header files that are shared between the compiler and the runtime. The epicsShare stuff has been removed from seq_prim_types.h. A new preprocessor symbol (declare_prim_type_names) makes sure that the prim_type_name and prim_type_tag_name variables are visible only inside the runtime library and the compiler and not in generated sequencer programs.

test:

- Set PATH for running tests in shared build

  The path to seq.dll needs to be set in cygwin and Windows for running tests when built with SHARED_LIBRARIES=YES

- Fix running tests on Windows

  The *Ioc.t tests were hanging on windows when run from Make - they worked when ran individually from the command line, but were not then killing the background IOC process on test completion. Using the Win32::Process package rather than fork() to create subprocesses fixes both issues

- set environment variables in a more portable way
- silence warnings about different TOP dirs
- ensure that tests cover seq exit phase
- Reorder test linking for cygwin

  Move subThreadSleep.c to separate library to enable correct import/export declarations for linking on cygwin

- enable pvSync with or without db
- set HARNESS_ACTIVE env var in queueTest.plt
- use a per host "unique" CA server port

  The idea here is to isolate concurrent test runs on the same machine against each other. The port is set to 10000 + pid % 30000, which is not unique in a strict sense, but the probability of a collision is quite small and the solution is non-intrusive and very simple to implement.

examples/demo:

- link specific code into a library instead of main

  This fixes problems for cygwin-x86 arch.

build system:

- added include of meta build configuration to configure/RELEASE
- removed configure/RELEASE.win32-x86

9.4 Release 2.2.2

The bugs listed in *Known Problems in Release 2.2.1* have been fixed.

In addition, some of the tests and parts of the test infrastructure have been changed so as to produce clean TAP files (this is currently only supported by base-3.15 with 'make tapfiles'). For the same reason, the usual messages when a program starts are no longer issued with printf, but rather with errlogSevPrintf(errlogInfo,...).

Some improvements to the web pages have been made: the version the pages refer to is more prominently visible, some shortcuts have been renamed, and side-bar shortcuts to all other (maintained) versions were added.

I added git mirrors of the darcs repos, the URLs are:

http://www-csr.bessy.de/control/SoftDist/sequencer/repo/branch-2-1.git http://www-csr.bessy.de/control/SoftDist/sequencer/repo/branch-2-2.git http://www-csr.bessy.de/control/SoftDist/sequencer/repo/branch-2-3.git

9.5 Release 2.2.1

This is the first release of version 2.2 of the sequencer. The following describes what has changed relative to version 2.1.

9.5.1 Supported EPICS Base Versions

This version drops support for base versions older than 3.14.12.2, all others should work fine.

If you manage to compile the sequencer with earlier base versions, well, good for you. It'll probably work, too. I will also accept patches that restore lost compatibility with older base versions, provided they achieve this without unduly complicating the code base or build system.

9.5.2 New Language Features

Foreign Types

It is now possible to use foreign types (i.e. types declared in C code, such as structs, enums, unions, and typedefs) in variable declarations, type casts, and the special "sizeof" built-in operator. Previously existing restrictions as to which type expressions are allowed have been lifted, so that (almost) everything you can say in C is now supported in SNL, too. Particularly, function types are now supported in declarations and type expressions, as well as indirect calls in expressions.

There are two notable limitations:

- Using C *type aliases* (defined in C with `typedef`) is allowed, but you must prefix the type name with the new keyword `typename` (which I borrowed from C++). Supporting foreign typedefs without any extra markup as in C would require lots of extra effort which IMO is not justified.
- There is only limited support for *defining* your own types (see Type Definitions below) and you cannot mix type definition with type usage (for instance in a variable declaration) as in C.

The words `enum`, `struct`, `typename`, `union`, and `void` are reserved words now and can no longer be used as identifiers.

Other Foreign Entities

A side effect of allowing indirect calls is that there is no longer any *syntactic* distinction between using functions and using variables. This means that issuing warnings just for the use of undeclared foreign *variables* is no longer possible. If they are still undeclared in the generated C code, the C compiler will warn you anyway. You can enable extra warnings (`+W`, see New Option for Extra Warnings below) to get these warnings, but that will report foreign functions, too.

A related change is that struct or union members are no longer identified with variables in the SNL syntax; members are no longer considered to be foreign entities (but for compatibility you can still list them in a foreign declaration).

Function Definitions

You can now define functions in SNL. The syntax is like in C and you *can* call the built-in PV functions inside them. This is made possible by passing the execution context (the state set identifier and the variable block pointer) as hidden parameters.

You may pass all kinds of variables to such a function, but for channel ("assigned") variables their special "assigned" status gets lost when passing them, similar as when passing them to a C function. This means you can call e.g. pvGet(x) in the body of an SNL function, but only if x is a global variable.

My various attempts to lift this limitation were the main reason it took me so much longer than I had expected to make this release. In order not to hold up the release of version 2.2 any longer, I decided to postpone this feature to a future release.

Type Definitions

You can now define your own struct types in SNL (other type formers like union and enum are not yet supported). As usual the syntax is a (simplified: no bit fields) variant of the C syntax. This is currently not very useful, since there is no way to "assign" struct members to PVs. Lifting this limitation is closely related to passing channel arguments to functions and therefore postponed, too.

New Option for Extra Warnings

The new option:`+W` (off by default) enables extra warnings, that you normally don't want to see. Currently this warns you once for each foreign entity that appears in the program.

9.5.3 Deprecated and Removed Features

Deprecated Foreign Declarations

The so called "foreign entity declarations", introduced in version 2.1, have become obsolete (see Other Foreign Entities) and are therefore *deprecated*.

Deprecated State Local `assign` etc

Using an `assign`, `monitor`, `sync`, or `syncq` clause inside a state is now deprecated. This was never very useful to begin with, and gets in the way of some future improvements.

Removed Keyword "connect"

This alias for the "assign" keyword, also introduced in version 2.1, is no longer supported. I believe nobody actually uses it, and the next version will introduce a completely new syntax for that feature anyway. I can demote the removal to a deprecation warning if it turns out that this seriously hurts people.

With hindsight, introducing this was a bad idea, as were foreign declarations and state local assign.

Removed PV Library

The PV library has been almost completely eliminated. What remains is a thin layer over CA, implemented in C, and offering only the functionality that is actually needed by the sequencer. The API is similar (but not identical) to the old C API; particularly, all the pvStat, pvSevr, and pvType definitions are as before.

The documentation for the PV layer has been removed, too. The only remaining user relevant part of the interface is contained in pvAlarm.h, see Built-in Constants.

Also removed were the Keck examples and the KTL related stuff in other example directories.

Removed devSequencer

The (broken and ugly) sequencer device support was removed. Consequently, seqFindProgByName was be removed as it is no longer needed.

9.5.4 Built-in Constants

All built-in constants are now known to `snc` and therefore no longer treated as foreign entities. Particularly, using them no longer gives "undefined variable" warnings, even with extra warnings enabled (`+W`). They are also documented in the reference.

The *snc*-generated C code for built-in constants now uses the name of the constant, instead of its value. This makes the code a bit more readable and slightly simplifies code generation.

The variables `ssId` and `pVar` are (somewhat irregularly) treated as built-in constants, too. This makes it possible to call C functions that take them as a parameter directly from inside SNL code without having to escape the call. That is, you can now write such calls as

```
my_c_fun(ssId, pVar, ...);
```

instead of

```
%%my_c_fun(ssId, pVar, ...);
```

9.5.5 Built-in Functions

Reported Severity of Failures

The severity of timeouts and usage errors reported on the console has been demoted from `errlogFatal` to `errlogMajor`. Reporting them as fatal has misled users in the past to think that their running program instances have become unstable now, which is definitely not the case for this sort of errors.

pvMessage

The built-in function `pvMessage` now consistently returns a failure string describing the problem whenever anything with the passed channel variable went wrong. This particularly applies to any failure status returned from the CA layer.

New Delay Implementation

While it was always *allowed* to use arbitrary SNL expressions for the argument to `delay`, even expressions that could change their value at any time (e.g. because they contain monitored variables), this did *not* work as one would expect. In fact, the delay expression would be evaluated just once for all delays appearing in conditions inside a state when the state was entered. Later checks whether the delay has expired used the cached delay value.

In the new (much simpler) implementation, delay expressions are evaluated like all other parts of the state transition condition. Delay IDs no longer exist and the code generator treats calls to delay like any other built-in function. The effect of delaying the state transition is now achieved (completely internal to the implementation of seq_delay) by adjusting the minimum time to wait for events (if the delay has not yet expired). The run-time system no longer stores the delay, but rather the (future) time when the wake-up should happen, leading to more accurate timing of delays.

Since calling delay outside of the condition of a state transition never had any useful effect, it is now disallowed.

Note: The prototype of the underlying C function seq_delay() has changed to support the new implementation. If there is code out there which calls this function from the C side, I'd venture that it is broken anyway and should be fixed.

---

pvAssignSubst

Unlike the `assign` clause, the dynamic `pvAssign` function does not substitute program parameters (previously sometimes referred to as "macros") in the channel name. There is now a new function `pvAssignSubst` that

behaves exactly like `pvAssign` except that it substitutes program parameters in the channel name, just like `assign` does.

PV Functions and Multiply Assigned Arrays

A long time wart of the sequencer, inherited from the 2.0 version, was the behaviour of built-in PV functions when you pass an array in which elements are assigned to separate PVs. In most cases (except one, see below) this behaved as if you had passed in just the first element of the array, which is inconsistent and error-prone.

However, existing programs might (perhaps without the author or maintainer being aware of it) rely on the current behaviour – and silently breaking such programs seems like a bad idea. Instead, this is now a *compile time error*. The error message contains a hint how the user can fix their program ("perhaps you meant to pass xxx[0]?") so that it retains its old behaviour.

For some of the PV functions, operating on all contained PVs of a multi-PV array would make sense if one would be willing to overload them. I decided against that and instead provide a number of new functions named "pvArray...":

- `pvArrayGetComplete`
- `pvArrayMonitor`
- `pvArrayPutComplete`
- `pvArrayStopMonitor`
- `pvArraySync`
- `pvArrayConnected`

I may add `pvArrayGet` and `pvArrayPut` in the future. Especially for the `SYNC` variant, such functions could be implemented (much) more efficiently than the semantically equivalent loop over all elements of the array.

Note: The C side equivalents seq_pvArray... are not (yet) part of the public API. The reason is that in version 2.3 they will get a slightly different type. You can use them in embedded C code but if you do that you should be aware that your program might not work or even build with version 2.3.

Timeout Arguments for pvGet and pvPut

For both `pvGet` and `pvPut` there is now the possibility to specify a timeout that differs from the default of 10 seconds. This is done by giving an extra argument after the `SYNC` keyword, as in:

```
pvGet(var,SYNC,1.0);
```

As before, the default behaviour for `pvGet(var)` i.e. neither an explicit `SYNC` nor `ASYNC`, is synchronous, unless option `-a` is in effect. In this case, or if `SYNC` is given with no extra argument, the standard default timeout of 10 seconds is assumed.

In contrast to previous releases of version 2.2, the C equivalents are unchanged with respect to version 2.1. Instead, the compiler generates calls to `seq_pvGetTmo` or `seq_pvPutTmo` which have the additional argument.

New Built-in Functions pvGetCancel and pvPutCancel

It is now possible to cancel and asynchronous get or put request by calling `pvGetCancel` or `pvPutCancel`, or the channel array variants `pvArrayGetCancel` or `pvArrayPutCancel`, respectively.

9.5.6 Generated C Code

Order of Definitions

The first change here is that the generated variable block is now placed *after* all other top-level definitions that appear in the program before the first state set, in particular before escaped C code. In previous versions, this was the other way around. The second change is that for all other top-level definitions the order is retained exactly as in the SNL source file.

This was done so that foreign type declarations can be used in global (SNL) variable declarations. This concerns types imported via `%%#include` as well as types defined in escaped C code.

Note that this means that escaped C code that appears before the first state set cannot access global variables declared in SNL, even if the re-entrant option is not in effect. Such C code should now be placed after the last state set.

Anyway, it is my hope that the possibility to write functions directly in SNL will make the escape-to-C route mostly obsolete.

Names of Generated Entities

These now follow a strict naming convention:

> Generated names start with `seqg_`.

In particular, `struct UserVar` became `struct seqg_var`, and the implicit parameters `ssId` and `pVar` are now named `seqg_ss` and `seqg_var`, respectively. Because these three identifiers are often referenced in escaped C code, I have added compatibility aliases for them, so as not to break existing programs. See External API below for details.

In order to prevent name clashes, words starting with `seqg_` are no longer valid identifiers in SNL.

I am aware that there may be cases where this change breaks existing programs which heavily rely on escape to C code. If this is the case, please shout! Adding more compatibility aliases can be done any time.

Event Flags

The compiler now generates variables for event flags to make it easier for escaped C code to call event flag functions (seq_ef*).

9.5.7 External API

I finally got around to separating the public API (to be called from e.g. embedded C code) from the internal interface between the compiler generated code and the run-time system. The latter parts have been moved from seqCom.h to the new header file seq_snc.h.

In particular, the following changes have been made:

Retained

- includes of pvAlarm.h and seq_release.h
- enum compType and its members DEFAULT, ASYNC, SYNC
- constants NOEVFLAG and DEFAULT_QUEUE_SIZE
- typedefs string, SS_ID, seqBool, seqProgram
- all functions except seqRegisterSequencerProgram and seqRegisterSequencerCommands but including seq_pvIndex (which is actually a macro)

Added

- new built-in functions:
  - seq_pvGetCancel
  - seq_pvPutCancel
  - seq_pvAssignSubst
- the constant DEFAULT_TIMEOUT (see New Delay Implementation)

Renamed

- EV_ID -> EF_ID
- VAR_ID -> CH_ID

Compatibility Aliases

```
typedef SEQ_VARS USER_VAR
#define ssId                    seqg_ss
#define pVar                    seqg_var
#define USER_VAR                SEQ_VARS
#define UserVar                 seqg_vars
#define VAR_ID                  CH_ID
#define EV_ID                   EF_ID
#define seq_pvFreeQ             seq_pvFlushQ
#define DEFAULT_QUEUE_SIZE      100
```

These names should not be used in new code.

Removed

- DELAY_ID (obsolete, see New Delay Implementation)
- OPT_MAIN (-m is a compile time option)
- optTest (is now internal to seq library)

Moved to seq_snc.h

- struct definitions for the static part of the generated program:
  - struct seqChan
  - struct seqState
  - struct seqSS
  - struct seqProgram
  - typedef PROG_ID
- option constants:
  - OPT_DEBUG
  - OPT_ASYNC
  - OPT_CONN
  - OPT_REENT
  - OPT_NEWEF

  - OPT_SAFE
  - OPT_NORESETTIMERS
  - OPT_DOENTRYFROMSELF
  - OPT_DOEXITTOSELF

  User code never needs to use these. It can use seq_optGet instead.

- bitmask operations:
  - NBITS
  - NWORDS
  - bitSet, bitClear, bitTest

  Like option values, these were never meant to be part of a public API.

- the constants TRUE and FALSE
- typedefs for generated functions; note that these have been renamed, too
- the functions seqRegisterSequencerProgram and seqRegisterSequencerCommands

Other Changes

- epicsShareAPI markers have been removed except for shell commands

Remarks

My justification for making these potentially breaking changes is that this is how it should have been done from the start. The things I have removed were never meant to be part of a public API. In the unlikely event that there are existing programs with a legitimate need to access these internals, they can always include seq_snc.h.

9.5.8 Build System

Top-level Makefile

The extra rules in the top level Makefile to produce documentation with or without generating a pdf file have been changed: 'pdf' and 'docs' are no longer make variables. Instead there are a number of new targets you can specify:

- html: build the docs in html format
- docs: additionally build the manual in pdf format
- upload etc: these are for my own benefit only, ignore them

See *Building the Manual* for details.

Generate st.d Files

The extra build rules for the sequencer have been extended to generate dependencies for .st files, using the mkmf.pl tool from base. Note that .st files are passed through the C preprocessor and therefore may include C header files or in fact SNL code from another file.

9.5.9 Bug Fixes

- Place program lock around wake-up commands in CA callback.

  This prevents a race condition resulting in a crash when the program shuts down and deletes mutexes etc before the callback has issued its final ss_wakeup call.

- Fixed connect and monitor accounting in seq_disconnect and seq_camonitor.

CHAPTER

TEN

KNOWN PROBLEMS

10.1 Known Problems in Release 2.2.5

- While building against base 7.x succeeds there are lots of deprecation warnings and most of the tests are not run due to an incomplete definition of the EPICS_HAS_UNIT_TEST make variable.

10.2 Known Problems in Release 2.2.4

- Building fails with Visual Studio 2010 due to a C99-ism (mixed declarations and code) that was introduced when upgrading the lemon parser template.
- Parallel building can fail when using gcc to generate dependency information. The solution is to use the EPICS tool mkmf.pl instead by placing a line

  HDEPENDS_METHOD = MKMF

  in the config section of src/snc/Makefile.
- On Windows, `make clean` fails inside the documentation directory. The fix is to replace `rm -rf` with `$(RMDIR)` in documentation/Makefile.

10.3 Known Problems in Release 2.2.3

- Certain versions of Windows don't like relative paths with forward slashes for commands whose result gets redirected to a file. This leads to errors when building against base-3.15 which has deprecated the `PATH_FILTER` function (which converts forward to backward slashes). The solution is to define `LEMON=$(abspath $(INSTALL_HOST_BIN)/lemon$(HOSTEXE))` in `src/snc/Makefile`.

10.4 Known Problems in Release 2.2.2

- Attempting to issue a DEFAULT pvPut when an ASYNC put is already pending on the same variable leads to an assertion failure.

10.5 Known Problems in Release 2.2.1

- In test/validate, pvAssignStress.st has the same program name as pvAssignSubst.st, leading to build errors when cross-compiling to VxWorks or RTEMS, where all tests get linked into a single library. This can be fixed by changing the program name in pvAssignStress.st to "pvAssignStressTest".
- A previous bugfix in the pvAssign builtin introduced a regression, that can lead to assertion failures.

- A wrong assertion in the CA event handler may be triggered when connection to a PV is lost in the middle of a pending get request.
- When exiting from a program, deadlock is possible when a pending callback is active when the program shuts down.
- Tests fail in tests/compiler if the build host has an ancient version of Test::More that does not support subtest. This happens e.g. on RHEL 6.6.
- Dependency generation in src/snc is broken if compiled against base-3.15 due to a bug in the Makefile (and one in base).
- Building static libraries fails due to wrong order of dependent libraries in two Makefiles.
- Building on Windows can fail due to missing epicsShare stuff in seq_snc.h.

CHAPTER

ELEVEN

PLANS FOR THE FUTURE

This is just a simple bullet list of things that are on my (long term) TODO list. Comments are welcome, as usual.

- Check whether it makes sense to let users specify the communication (request) type. This is currently baked-in to be inferred from the variable's type as specified in the program text. Another twist would be to allow to use the native type and convert on the seq side. The question is if this brings any actual benefit, considering that sequencer and database typically run on the same IOC. It might be convenient, sometimes.
- Add "pv" as a prefix type constructor. This is now implemented in my working branch for version 2.3. Sketch:

```
int pv x = 0;                          /* single anonymous scalar pv */
int pv x = pv "name";                  /* single named scalar pv */
int (pv x)[2] = {1,2};                 /* single anonymous array valued pv */
int (pv x)[2] = pv "name";             /* single named array valued pv */
int pv x[2];                           /* array of 2 anonymous scalar pvs */
int pv x[2] = pv {"n1", "n2"};         /* array of 2 named scalar pvs */
int (pv x[2])[100];                    /* array of 2 anonymous array valued pvs */
int (pv x[2])[100] = pv {"n1", "n2"}; /* array of 2 named array valued pvs */
```

  This now also works for structs whose members can be of pv type etc.

- Raw ideas:
  - Allow parameterized state sets (like a function). Then allow to "jump" from one state set to another one. The "caller" i.e. the original state set must wait for the "callee" to finish. The idea here is to avoid dynamic creation of threads, as this could be implemented as a simple procedure call. Needs more thought.
  - Better support for enumerations i.e. PVs with native request type DBR_ENUM. The idea is that the programmer can use identifiers to name the choices, but does not have to know in advance which integer value corresponds to which choice.

    Maybe something very simple suffices, like a built-in function

    ```
    int pvChoice(var, char *choice_name);
    ```

    This requires support for DBR_GR_ENUM in the pv layer.
  - Better syntax for assign, monitor, sync, etc. The idea is to provide an alternative to the usual macros. Every SNL programmer uses her own self-made macros to simplify declaration blocks like

    ```
    int x;
    assign x to "bla";
    monitor x;
    sync x to ef_x;
    ```

    The challenge here is to come up with something that is light-weight, can be easily specialized, and seamlessly fits into the existing syntax.

    My current favorite candidate is to extend the pv initialization syntax. The idea is that a pv init expression (i.e. an initializer prefixed with the "pv" keyword) can get extra arguments with defaults:

```
int pv x = pv "name";              /* just assigned to "name" */
int pv x = pv ("name");            /* same */
int pv x = pv ("name", 1);         /* also monitor */
int pv x = pv ("name", 1, ef);     /* also sync to ef */
int pv x = pv ("name", 0, ef);     /* no monitor, but sync to ef */
int pv x = pv ("name", 1, 0, 20); /* monitor, no sync, queued (size 20) */
int pv x[2] = pv ({"n1", "n2"},1);/* both pvs are monitored */
struct two {
  int pv x;
  int pv ys[2];
};
struct two s = pv {"n1", {"n2", "n3"}}; /* no monitors */
struct two s = pv {("n1",1), ({"n2", "n3"}, 1, ef)}; /* x monitored, ys also synced */
etc...
```

The "pv" outside of an aggregate "{ ... }" should be the same as prefixing all elements with it. Aggregates in a pv argument position are allowed only for the pv name and mean expand all elements with the other arguments.

- Convert everything under test to use epicsUnitTest. Add more regression tests.
- Need to go over the introductory chapters in the docs, remove out-dated information. Partly done.
- Götz suggested to allow the monitor clause inside a state set, even if the variable is global. This would mean automatic variable updates affect only those state sets which contain the monitor clause, while others do not get updates. (Perhaps: combine this with state set local event flags and sync clauses.)

  This could be useful in situations where one state set writes to the variable and others only read. The writer would not monitor, thus would not fall victim to the modify but not publish fallacy.

  Experimental implementation in version 2.3 branch done. State local assign, monitor, etc got in the way, which is why I removed them in 2.3 (deprecated in 2.2). They aren't too useful anyway.

- Add some sort of built-in macro language. Something that nicely integrates with the existing SNL syntax would be nice, e.g.

  ```
  define macro (arg, ...) {
    definition
  }
  include "headerfile";
  ```

  Use of CPP could then be deprecated. What about token catenation and stringification as in CPP?

  Implementation would have to be an intermediate step between lexing and parsing.

CHAPTER

TWELVE

GLOSSARY OF TERMS

sequencer

1. The project that defines SNL and provides an implementation.
2. The runtime library that supports execution of SNL programs.

cpp The C preprocessor. Normally not a stand-alone program but part of the C compiler. Can be abused to preprocess SNL code, too, which is a constant source of compatibility problems.

snc The SNL compiler. See *Compiling SNL Programs*.

SNL State Notation Language. What this project is about.

assigned The property of a variable or array element to occur in an assign clause in an SNL program.

PV, process variable A mutable variable that is globally visible and accessible via some network protocol in a distributed control system, such as one based on EPICS.

CA, Channel Access The EPICS network protocol, used to connect client programs to PVs hosted on a server.

EPICS Experimental Physical and Industrial Control System. A set of tools and libraries for building large distributed soft real-time control systems.

state set SNL jargon for finite state machine. A program may contain multiple state sets and each runs in its own thread.

true Any non-zero value, as in C.

false Zero, as in C.

- Index

Symbols

+W
command line option, 27
+a
command line option, 26
+c
command line option, 26
+d
command line option, 26
+e
command line option, 26
+i
command line option, 26
+l
command line option, 26
+m
command line option, 26
+r
command line option, 26
+s
command line option, 26
+w
command line option, 26
-W
command line option, 27
-a
command line option, 26
-c
command line option, 26
-d
command line option, 26
-e
command line option, 26
-i
command line option, 26
-l
command line option, 26
-m
command line option, 26
-o
command line option, 26
-r
command line option, 26
-s
command line option, 26
-w
command line option, 26

A

assigned, **93**

C

CA, **93**
channel (C type), 60
Channel Access, **93**
command line option
+W, 27
+a, 26
+c, 26
+d, 26
+e, 26
+i, 26
+l, 26
+m, 26
+r, 26
+s, 26
+w, 26
-W, 27
-a, 26
-c, 26
-d, 26
-e, 26
-i, 26
-l, 26
-m, 26
-o, 26
-r, 26
-s, 26
-w, 26
compType (C type), 59
cpp, **93**

D

delay (C function), 60

E

efClear (C function), 68
efSet (C function), 68
efTest (C function), 68
efTestAndClear (C function), 68
EPICS, **93**

F

false, **93**

M

macValueGet (C function), 68

N

NOEVFLAG (C macro), 59

O

optGet (C function), 69

P

process variable, **93**
PV, **93**
pvArrayConnected (C function), 67
pvArrayGetCancel (C function), 63
pvArrayGetComplete (C function), 62
pvArrayMonitor (C function), 65
pvArrayPutCancel (C function), 62
pvArrayPutComplete (C function), 61
pvArrayStopMonitor (C function), 65
pvArraySync (C function), 65
pvAssign (C function), 64
pvAssignCount (C function), 67
pvAssigned (C function), 66
pvAssignSubst (C function), 64
pvChannelCount (C function), 67
pvConnectCount (C function), 68
pvConnected (C function), 67
pvCount (C function), 65
pvFlush (C function), 67
pvFlushQ (C function), 64
pvFreeQ (C function), 63
pvGet (C function), 62
pvGetCancel (C function), 63
pvGetComplete (C function), 62
pvGetQ (C function), 63
pvIndex (C function), 67
pvMessage (C function), 66
pvMonitor (C function), 64
pvPut (C function), 60
pvPutCancel (C function), 61
pvPutComplete (C function), 61
pvSeverity (C function), 66
pvSevr (C type), 59
pvStat (C type), 58
pvStatus (C function), 66
pvStopMonitor (C function), 65
pvSync (C function), 65
pvTimeStamp (C function), 66

S

seq (C function), 33
seqBool (C type), 59
seqcar (C function), 35
seqChanShow (C function), 34
seqQueueShow (C function), 35
seqShow (C function), 33
seqStop (C function), 36
sequencer, **93**
snc, **93**
SNL, **93**
state set, **93**

T

true, **93**