State Notation Language and Sequencer Users Guide

Version 2.0
(for EPICS release 3.13 and later)

Manual Revision 1.9 (DRAFT)
February 18, 1998
Written by Andy Kozubal

Instrumentation and Control Group
Dynamic Experimentation Division
Mail Stop P942
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Phone: (505) 667-6508
EMAIL: AKozubal@LANL.Gov

Manual Revision 2.0 (DRAFT 3)
February15, 2000
Updated by William Lupton
(with some material by Greg White)

W. M. Keck Observatory
65-1120 Mamalahoa Highway
Kamuela, Hawaii 96743

Phone: (808) 885 7887
EMAIL: WLupton@Keck.Hawaii.Edu

1. Introduction

Note on Versions

Version 1.9 of this manual described version 1.9 of the sequencer and was written by Andy Kozubal, the original author of this software. This version of the manual describes versions 1.9.5 and 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).

Version 2.0 differs from version 1.9.5 mainly in that sequencer run-time code is once again portable between VxWorks and Unix, and message systems other than channel access can be used. Version 2.0 differences are always noted. Version 2.0 can not be used with EPICS R3.13; it is dependent on libraries which will be generally available only with EPICS R3.14.

Version 1.9.5 is being made available to the EPICS community but all new developments apart from major bug fixes will be based on version 2.0.

Overview

The state notation language (SNL) provides a simple yet powerful tool for programming sequential operations in a real-time control system. Based on the familiar state-transition diagram concepts, programs can be written without the usual complexity involved with task scheduling, event handling, and input/output programming.

Programs produced by the state notation language are executed within the framework of the run-time sequencer. The sequencer drives the program to states based on events, and establishes interfaces to the program that enable it to perform real-time control in a multi-tasking environment. The sequencer also provides services to the program such as establishing connections to run-time database channels and handling asynchronous events.

The state notation language and sequencer are components of the Experimental Physics and Industrial Controls System (EPICS). EPICS is a system of interactive applications development tools (toolkit) and a common run-time environment (CORE) that allows users to build and execute real-time control and data acquisition systems for experimental facilities, such as particle accelerators and free electron lasers. EPICS is a product of the Accelerator Automation and Controls Group (AOT-8), which is within the Accelerator Operations and Technology (AOT) Division at the Los Alamos National Laboratory. The sequencer interfaces to the run-time database through the channel access facility of EPICS.

Content of this Manual

This users manual describes how to use the state notation language to program real-time applications. The user is first introduced to the state notation language concepts through the state-transition diagram. Through a series of examples, the user gains an understanding of most of the SNL language elements. Next, the manual explains procedures for compiling and executing programs that are generated by the SNL. Testing and debugging techniques are presented. Finally, we present a complete description of the SNL syntax and the sequencer options.

Omissions from this manual

This manual should contain more information on the following subjects:

- future plans
- the PV (process variable) API
- real-life annotated examples of **`assign`**, **`sync`**, **`syncQ`**

Copyright and Restrictions

Notes on This Release

New version 1.9 features have been moved to “New features in Version 1.9” on page 41. This section gives brief notes on new version 1.9.5 and version 2.0 changes.

Version 1.9.5 of the sequencer and state notation compiler is available for EPICS release 3.13 and later. We have added several enhancements to the language and to the run-time sequencer. State programs must be compiled under the new state notation compiler to execute properly with the new sequencer. However, no source-level changes to existing programs are required.

Version 1.9.5 differs from version 1.9.4 only in that sequencer deletion from the VxWorks shell once again works.

New Language Features

Entry and exit blocks

The **`entry{}`** block of a state is executed each time the state is entered; the **`exit{}`** block is executed each time the state is left.

State options

`-t`, **`-e`** and **`-x`** are now recognized options within the scope of a state. **`-t`** inhibits the “timer reset” on re-entry to a state from itself; **`-e`** (for “entry”) is used with the new **`entry{}`** block, and forces the **`entry{}`** statements to be executed on all entries to a state, even if from the same state; **`-x`** (for “exit”) is complimentary to **`-e`**, but for the new **`exit{}`** block.

Queueable monitors

Monitor messages can be queued and then dequeued at leisure. This means that monitor messages are not lost, even when posted rapidly in succession. This feature is supported by new **syncQ**, **pvGetQ** and **pvFreeQ** language elements, and a new **seqQueueShow** routine.

efClear can wake up state sets

Clearing an event flag can now wake up state sets which reference the event flag in **when** tests.

More C syntax is supported

Compound expressions such as **i=1,j=2** (often used in for loops) are now permitted.

Variables can now be initialized in declarations such as **int i=2;**.

Pre-processor “**#**” lines are now permitted between state sets and states.

“**~**” (complement) and “**^**” (exclusive or) operators are permitted.

ANSI string concatenation, e.g. **"xxx" "yyy"** is the same as **"xxxyyy"**, is supported.

Full exponential representation is supported for numbers (previously couldn’t use “**E**” format).

Bugs fixed

Avoidance of segmentation violations

- SEGV no longer occurs if an undeclared variable or event flag is referenced
- SEGV no longer occurs if the last bit of an event mask is used

Avoidance of race condition which prevented monitors from being enabled

If a connection handler was called before **seq_pvMonitor**, a race condition meant that the **ca_add_array_event** routine might never get called.

Miscellaneous

Compilation warnings have been avoided wherever possible.

A 60Hz system clock frequency is no longer assumed.

Version 2.0 changes

Replaced VxWorks dependencies with OSI routines

All VxWorks routines have been replaced with the appropriate OSI (Operating System Independent) routines.

Some VxWorks routines do not yet have OSI equivalents and have been commented out (**taskwd**, log file support, **taskIdFigure**, task delete hooks and sequencer deletion).

Unused (and undocumented) **`VX_OPT`** option has been removed.

Replaced direct channel access calls with new PV API

All CA calls have been replaced with equivalent calls to a new PV (process variable) API which can be layered on top of not just CA but also other message systems.

Added optional generation of main program

The new **`+m`** (main) option generates a Unix main program whose single argument is a list of macro assignments.

Under Unix, the main thread reads from standard input and can execute **`seqShow`**, **`seqChanShow`** etc. on demand. EOF causes the sequencer to exit.

Fixed more bugs

Several minor (and long-standing) bugs were found while testing with Purify, e.g. a NULL pointer dereference and reading outside the bounds of a macro value string.

Improved error reporting

Error reporting is now more consistent. It is currently just using **`epicsPrintf`**.

2. State Notation Language Concepts

The State Transition Diagram

The state transition diagram is a graphical notation for specifying the behavior of a control system in terms of control transformations. The state transition diagram or STD serves to represent the action taken by the control system in response to both the present internal state and some external event or condition. To understand the state notation language one must first understand the STD schema.

A simple STD is shown in figure 1. In this example the level of an input voltage is sensed, and a light is turned on if the voltage is greater than 5 volts and turned off if the voltage becomes less than 3 volts. Note that the output or action depends not only on the input or condition, but also by the current memory or state. For instance, specifying an input of 4.2 volts does not directly specify the output, but depends on the current state.

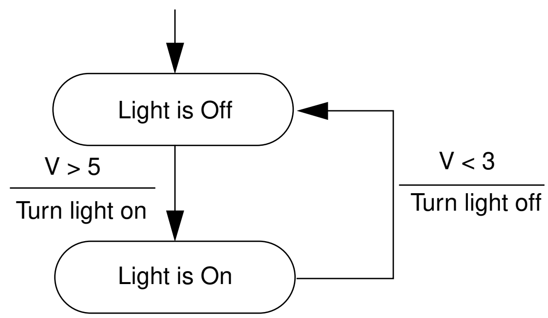

Figure 1: A simple state transition diagram

Elements of the State Notation Language

The following SNL code segment expresses the state transition diagram in figure 1:

```
state light_off {
   when (v > 5.0){
      light = TRUE;
      pvPut(light);
   } state light_on
}

state light_on {
   when (v < 3.0){
      light = FALSE;
      pvPut(light);
   } state light_off
}
```

You will notice that the SNL appears to have a structure and syntax that is similar to the C language. In fact the SNL uses its own syntax plus a subset of C, such as expressions, assignment statements, and function calls. This example contains two code blocks that define states: **`light_off`** and **`light_on`**. Within these blocks are **`when`** statements that define the events ("v > 5.0" and "v < 3.0"). Following these statements are blocks containing actions (C statements). The **`pvPut`** function writes or puts the value in the variable **`light`** to the appropriate database channels. Finally, the next states are specified following the action blocks.

For the previous example to execute properly the variables **`v`** and **`light`** must be declared and associated with database channels using the following declarations:

```
float   v;
short   light;
assign  v to "Input_voltage";
assign  light to "Indicator_light";
```

The above ***assign*** statements associate the variables **`v`** and **`light`** with the database channels "Input_voltage" and "Indicator_light" respectively. We want the value of **`v`** to be updated from the database whenever it changes. This is accomplished with the following declaration:

```
monitor v;
```

Whenever the value of the database changes the value of **`v`** will likewise change (within the time constraints of the underlying system).

A Complete State Program

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

```
program level_check
float   v;
assign  v to "Input_voltage";
monitor v;
short   light;
assign  light to "Indicator_light";

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

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

To distinguish a state program from other state programs it must be assigned a name. This was done in the above example with the statement:

```
program level_check
```

As we'll see in the next example, we can have multiple state transition diagrams in one state program. In SNL terms these are referred to as *state sets.* Each state program may have one or more named state sets. This was denoted by the statement block:

```
ss volt_check { ... }
```

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). This output is the same channel that is used as input by the **volt_check** state set.

First, we add the following lines to the declaration:

```
float   vout;
float   delta;
assign  vout to “ts1:ai1”;
```

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 **when** statement in state **init** contains an empty event expression. This means unconditional execution of the transition. Because init is the first state in the state set, it is assumed to be the initial state. You will find this to be a convenient method for initialization. Also, notice that the **ramp** state always returns to itself. This is a permissible and often useful construction. The structure of this state set is shown in the STD in figure 2.

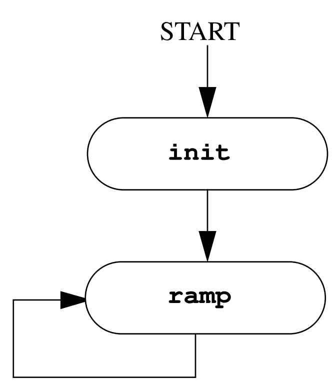

Figure 2: Structure of **generate_voltage** State

The final concept introduced in the last example is the **delay** function. This function returns a TRUE value after a specified time interval from when the state was entered. The parameter to **delay** specifies the number of seconds, and must be a floating point value (constant or expression).

At this point, you may wish to try an example with the two state sets. You can jump ahead and read parts of Sections 3-5. You probably want to pick a unique name for your database channels, rather than the ones used above. You may also wish to replace the **pvPut** statements with **printf** statements to display “High” and “Low” on your console.

Database Names Using Macros

One of the features of the SNL and run-time sequencer is the ability to specify the names of database channels at run-time. This is done by using macro substitution in the database name. In our example we could replace the **assign** statements with the following:

```
assign Vin to "{unit}:ai1";
assign Vout to "{unit}:ao1";
```

The string within the curly brackets is a macro which has a name (“unit” in this case). At run-time you give the macro a value, which is substituted in the above string to form a complete database name. For example, if the macro “unit” is given a name “DTL_6:CM_2”, then the run-time channel name is “DTL_6:CM_2:ai1”. More than one macro may be specified within a string, and the entire string may be a macro. See Section 4. on page 19 for more on macros.

Data Types

The allowable variable declaration types correspond to the C types: **char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long, unsigned long**, **float**, and **double**. In addition there is the type **string**, which is a fixed array size of type **char**. Variables having any of these types may be assigned to a database channel. The type declared does not have to be the same as the native database value type. The conversion between types is performed at run-time.

You may specify array variables as follows:

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

When assigned to a database channel the database operations, such as **pvPut**, are performed for the entire array.

Arrays of Channels

Often it is necessary to have several associated database channels. The ability to assign each element of an array to a separate channel 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:

```
assignVin 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;
```

Double subscripts offer additional options.

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

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

Dynamic Assignment

You may declare a variable and defer its assignment until later by assigning it to an empty string as follows:

```
float   Xmotor;
assign  Xmotor to ""; /* not assigned yet */
...
      /* dynamic assignment */
      pvAssign(Xmotor, "bpm04:motor_x");
```

Unassigned Channels

You may also de-assign a variable from a channel as follows:

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

The total number of assigned channels is returned by the function **`pvAssignCount`**:

```
NumAssigned = pvAssignCount();
```

Status of Database Channels

Most database record types have associated with them an alarm status and alarm severity. You can obtain the alarm status and severity with the **`pvStatus`** and **`pvSeverity`** functions. For example:

```
when (pvStatus(x_motor) != NO_ALARM) {
    printStatus("X motor",
        pvStatus(x_motor), pvSeverity(x_motor));
    ...
```

These routines are described in Section 5. on page 23. The values for alarm status and severity are defined in the EPICS include file **`alarm.h`**.

You can obtain the time stamp with the **`pvTimeStamp`** function. For example:

```
time = pvTimeStamp(x_motor);
```

Version 2.0: EPICS-specific types are no longer used. **`pvStatus`** will return an enumeration of type **`pvStat`**, **`pvSeverity`** will return an enumeration of type **`pvSevr`**, and **`pvTimeStamp`** will return a structure of type **`pvStamp`**. All of these are defined in the include file **`pv.h`** and are listed in xxx.

Synchronizing State Sets with Event Flags

State sets within a state 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 **`when`** clause. The **`sync`** statement may also be used to associate an event flag with a database channel that is being monitored. In that case whenever a monitor returns, the corresponding event flag is set. Note that this provides an alternative to testing the value of the monitored channel. This 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. See Section 6. on page 34 for an example using event flags.

Queuing Monitors

Neither testing the value of a monitored channel in a **`when`** 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 channel 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 new **`syncQ`** statement to associate a channel with a queue and an event flag. The new **`pvGetQ`** function retrieves and removes the head of queue. See xxx for an example using queued monitors.

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`** compile flag (see Section 3. on page 15). The operation may 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 Section 5. on page 23.

Connection Management

All database channel connections are handled by the sequencer through the channel access interface. Normally the state programs are not run until all database channels are connected. However, with the **`-c`** compile flag execution begins while the connections are being established. The program can test for each channel's connection status with the **`pvConnected`** routine, or it can test for all channels connected with the following comparison:

```
pvChannelCount() == pvConnectCount()
```

These routines are described in Section 5. on page 23. If a channel disconnects or re-connects during execution of a state program the sequencer updates the connection status appropriately.

Multiple Instances and Reentrant Object Code

Occasionally you will create a state program that can be used in multiple instances. If these instances are on separate processors, there is no problem. However, if more than one instance must be executed simultaneously on a single processor, then the objects must be made reentrant using the **`+r`** compile 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 all accesses to variables is slightly less efficient.

Database Variable Element Count

All database requests for database variables that are arrays assume the array size for the element count. However, if the database channel 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);
      ...
```

Dynamic Assignment

You may dynamically assign or re-assign variable to database channels 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);
```

An empty string in the assign declaration implies no initial 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()
```

3. Compiling a State Program

This section describes how to compile a state program in preparation for execution by the run-time sequencer. You should first consult the user manual "EPICS: Setting Up Your Environment".

The State Notation Compiler

The state notation compiler (SNC) converts the state notation language (SNL) into C code, which is then compiled to produce a run-time object module. The C pre-processor (cpp) may be used prior to the SNC. If we have a state program file named "test.st" then the steps to compile are similar to the following:

```
snc test.st
gcc -c test.c -O ...additional compile options
```

Alternatively, using the C pre-processor:

```
cpp test.st test.i
snc test.i
gcc -c test.c -O ...
```

Using the C pre-processor allows you to include SNL files (**`#include`** directive), to use **`#define`** directives, and to perform conditional compiling (e.g. **`#ifdef`**).

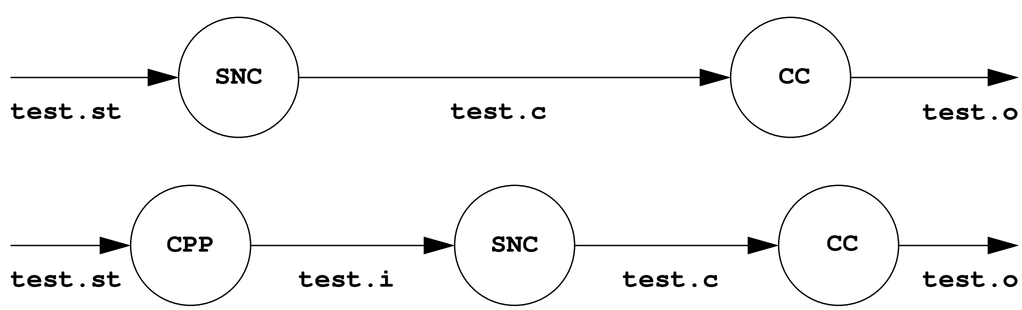

Figure 3: Two Methods of Compiling a State Program

Name of output file

The output file name will 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 that the above: **`.st`** and single-character extensions are replaced with **`.c`**; otherwise **`.c`** is appended to the full file name. In all cases, **`-o`** overrides.

Compiler Options

SNC provides 8 compiler options. You specify the option by specifying a key character preceded by a plus or minus sign. A plus sign turns the option on, and a minus turns the option off. The options are:

+a Asynchronous **pvGet**, i.e. the program will proceed before the operation is completed.

−a **pvGet** returns after the operation is completed. This is the default if an option is not specified.

+c Wait for all database connections before allowing the state program to begin execution. This is the default.

−c Allow the state 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).

+l Produce C compiler error messages with references to source (.st) lines. This is the default.

−l Produce C compiler error messages with references to .c file lines.

+m Generate a Unix C main program (a wrapper around a call to the **seq** function).

−m Do not produce a Unix C main program. This is the default.

+r Make the run-time code reentrant, thus allowing more than one instance of the state program to run on an IOC.

−r Run-time code is not reentrant, thus saving start-up time and memory. This is the default.

+w Display SNC warning messages. This is the default.

−w Suppress SNC warnings.

Options may also be included within the declaration section of a state program:

```
option   +r;
option   -c;
```

Cross Compilers and Makefiles

When the target architecture is different from the host's, a cross compiler must be used. We recommend setting up a **Makefile** to compile state programs.

The C file produced by SNC must be compiled with the include file **seqCom.h**, which contains many necessary definitions and declarations and which includes the EPICS **tsDefs.h** file. The **Makefile** should reference the director(ies) where these files are located.

Version 2.0: **seqCom.h** is no longer dependent on **tsDefs.h**.

Compiler Errors

The SNC detects most errors, displays an error message with the line number, and aborts further compilation. Some errors may not be detected until the C compilation phase. Such errors will display the line number of the SNL source file. If you wish to see the line number of the C file then you should use the **-l** (“ell”) compiler option. However, this is not recommended unless you are familiar with the C file format and its relation to the SNL file.

Warnings from SNC

Certain inconsistencies detected by the SNC are flagged with error messages. An example would be a variable that is used in the SNL context, but declared in escaped C code. These warnings may be suppressed with the **-w** compiler option.

Compiling and linking a state program under Unix

Under Unix, the **+m** compiler option should be used to create a C main program. The state program should then be compiled with the **_REENTRANT** macro defined, and linked against the following libraries: **Seq**, **Osi**, **ca** (if using channel access), **Com**. It may be necessary to search **Osi** again after **Com**. It is also necessary to search the Operating System’s thread library and possibly other libraries in order to resolve missing references.

For example, here is a full build of a simple state program from source under Solaris. It is assumed that the appropriate EPICS version is in **/usr/local/epics**.

```
cpp demo.st demo.i

snc +m demo.i

cc -D_REENTRANT \
   -I. -I.. -I../../include -I../../include/os/solaris \
   -I/usr/local/epics/base/include \
   -I/usr/local/epics/base/include/os/solaris \
   -c demo.c

CC -o demo demo.o \
   -L/usr/local/epics/base/lib/solaris \
   -L../../lib/solaris \
   -lSeq -lOsi -lca -lCom -lOsi -lpthread -lthread \
   -lsocket -lnsl -lposix4 -lm
```

The main program generated by the **+m** compiler option is very simple. Here it is:

```
int main(int argc,char *argv[]) {
    char *macro_def = (argc>1)?argv[1]:NULL;
return seq((void *)&demo, macro_def, 0);
}
```

The arguments are essentially the same as those taken by the **seq** routine. You can write your own if you want, e.g. to link multiple state programs into the same Unix executable (this has not been tested, but it should work).

4. Using the Run Time Sequencer

In the previous section you learned how to create and compile some simple state programs. In this section you will be introduced the run-time sequencer so that you can execute your state program. We assume you are familiar with the VxWorks environment.

Loading the sequencer

The sequencer is unbundled from EPICS base and so must be loaded separately. The sequencer is loaded into an IOC by the VxWorks loader from object files on the UNIX file system. Assuming the IOC's working directory is set properly, the following command will load the sequencer object code:

```
ld < seq
```

Loading a State Program

State programs are loaded into an IOC by the VxWorks loader from object files on the UNIX file system. Assuming the IOC's working directory is set properly, the following command will load the object file "example.o":

```
ld < example.o
```

This can be typed in from the console or put into a script file, such as the VxWorks start-up file.

Executing the State Program

Let's assume that the program name (from the **`program`** statement in the state program) is "level_check". Then to execute the program under VxWorks you would use the following command:

```
seq &level_check
```

This will create one task for each state set in the program. The task ID of the first state set task will be displayed. You can find out which tasks are running by using the VxWorks "**`i`**" command.

Under Unix, you execute the state program directly. You might type the following:

```
level_check
```

Deleting the State Program Tasks

Deleting any one of the state set tasks will cause all tasks associated with the state program to be deleted. For example, under VxWorks:

```
td "level_check"
```

A state program may delete itself. The suggested method is to place the following statement at an appropriate place within the program:

```
exit();
```

Under Unix, a state program may be killed by sending it a SIGTERM signal or entering an EOF character.

Specifying Run-Time Parameters

You can specify run-time parameters to the sequencer. Parameters serve three purposes: (1) macro substitution in process variable names, (2) for use by your state program, and (3) as special parameters to the sequencer. You can pass parameters to your state program at run time by including them in a string with the following format:

```
"param1 = value1, param2 = value2, ... "
```

For example, if we wish to specify the value of the macro “unit” in the example in the last chapter, we would execute the program with the following command:

```
seq &level_check, "unit=DTL_6:CM_2"
```

This works just the same under Unix. The above example becomes:

```
level_check "unit=DTL_6:CM_2"
```

Parameters can be accessed by your program with the function **`macValueGet`**, which is described in Section 5. on page 23. The following built-in parameters have special meaning to the sequencer:

`logfile =` ***filename***

This parameter specifies the name of the logging file for the run-time tasks associated with the state program. If none is specified then all log messages are written to the console (standard output under Unix).

`name =` ***task_name***

Normally the task names are derived from the program name. This parameter specifies an alternative base name for the run-time tasks.

`stack =` ***stack_size***

This parameter specifies the stack size in bytes (it’s ignored under Unix).

`priority =` ***task_priority***

This parameter specifies the initial task priority when the tasks are created. The value ***task_priority*** must be an integer between 1 and 255 (it’s ignored under Unix).

Examining the State Program

Under VxWorks, you can examine the state program by typing:

```
seqShow "level_check"
```

This will display information about each state set (e.g. state set names, current state, previous state). You can display information about the database channels associated with this state program by typing either of:

```
seqChanShow "level_check"
seqChanShow "level_check", "DTL_6:CM_2:ai1"
seqChanShow "level_check", "-"
```

You can display information about monitor queues by typing:

```
seqQueueShow "level_check"
```

The first parameter to **seqShow**, **seqChanShow** and **seqQueueShow** is either the task identifier (tid) or the task name of the state program task. If the state program has more than one tid or name, then any one of these can be used. The second parameter is a valid channel name, or "**-**" to show only those channels which are disconnected, or "**+**" to show only those channels which are connected. The **seqChanShow** and **seqQueueShow** utilities will prompt for input after showing the first or the specified channel; enter **RETURN** or a signed number to view more channels or queues; enter "**q**" to quit.

If you wish to see the task names, state set names, and task identifiers for *all* state programs type:

```
seqShow
```

Similar but shorter commands can be issued under Unix. Valid commands are **show**, **chan** and **queue**. They are abbreviable.

Sequencer Logging

The sequencer logs various information that could help a user determine the health of a state program. Logging goes to the console by default, but may be directed to any file by specifying the logfile parameter as described above.

What Triggers an Event?

The run-time sequencer uses four methods to test an event:

- a database value returns from database (monitor or **pvGet**)
- a time delay has elapsed
- an event flag is set or cleared
- any channels connect or disconnect

When one of these events occur, the sequencer executes the appropriate **when** statements based on the current states and the particular event or events. Whenever a new state is entered, the corresponding **when** statements for that state are executed immediately, regardless of the occurrence of any of the above events.

Prior to Version 1.8 of the sequencer the event flags were cleared after a when statement executed. Currently, event flags must be cleared with either **`efTestAndClear`** or **`efClear`**, unless the **`-e`** option was chosen.

5. State Notation Language Syntax

This section formalizes the state notation language syntax using a variant of BNF (Backus-Naur Form). The idea is that the meaning will be clear without explanation. However, here are some explanatory notes.

- words in `teletype` font are to be taken literally ("terminals")
- words in ***bold italics*** are syntactic terms which will be defined below ("nonterminals"), except in a few cases where the meaning is obvious
- where the name of a nonterminal begins with the word ***optional*** and it is enclosed in square brackets, that term is optional
- where a term is followed by an ellipsis (**...**), it may optionally be repeated (so if the term was not optional this means that there can be one or more instances of it; if the term was optional this means that there can be zero or more instances of it)
- where a term is followed by a separator (e.g. a comma) and an ellipsis, it is to be understood that the separator will separate each repeated instance of the term

State Program

A state program has the following structure:

`program` ***program_name*** `;`
declarations
`[`***state_set***`] ...`

The program name may be followed by a parameter list:

`program` ***program_name*** `("`***parameter_list***`");`

Declarations

Variable declarations are similar to C except that the types are limited to the following, no initialization is permitted, and only one variable may be declared per declaration statement.

`char` ***variable_name*** `;`
`short` ***variable_name*** `;`
`int` ***variable_name*** `;`
`long` ***variable_name*** `;`
`float` ***variable_name*** `;`
`double` ***variable_name*** `;`
`string` ***variable_name*** `;`

Type **`string`** produces an array of char with length equal to the constant **`MAX_STRING_SIZE`**, which is defined in one of the included header files. Unsigned types and pointer types may also be specified. For example:

`unsigned short` `*`***variable_name*** `;`

Variable may also be declared as arrays.

```
char    variable_name[array_length];
short   variable_name[array_length];
int     variable_name[array_length];
long    variable_name[array_length];
float   variable_name[array_length];
double  variable_name[array_length];
char    variable_name[array_length][array_length];
short   variable_name[array_length][array_length];
int     variable_name[array_length][array_length];
long    variable_name[array_length][array_length];
float   variable_name[array_length][array_length];
double  variable_name[array_length][array_length];
```

Note that we have not yet implemented arrays of strings.

Assignment of a Variable to a Database Channel

Once a variable is declared, it may be assigned to a database channel. Thereafter, that variable is used to perform database operations. All of the following are variations on assignment:

```
assign  variable_name to "database_name";
assign  variable_name[index] to "database_name";
assign  variable_name to { "database_name", ... };
```

A database name may contain one or more macro names enclosed in brackets: "`{ ... }`". Macros are named following the same rules as C language variables.

For database variable declared as arrays, the requested count is the length of the array or the native count for the database channel, whichever is smaller. The native count is determined when the initial connection is established. Pointer types may not be assigned to a database channel.

Monitoring a Database Channel

To make the state program event-driven the input variables can be monitored. Monitored variables are automatically updated with the current database value. The variable must first be assigned to a database channel.

```
monitor   db_variable_name;
monitor   db_variable_name[index];
```

Declaring Event Flags

Event flags are declared as follows:

```
evflag    event_flag_name;
```

Associating an Event Flag with a Database Channel

An event flag may be associated with a database channel. When a monitor returns on that channel the corresponding event flag is set.

sync ***variable_name*** ***event_flag_name*** **;**

Associating an Event Flag with a Queued Database Channel

An event flag may be associated with a queued database channel. The queue size defaults to 100 but can be overridden on a per-channel basis. When a monitor returns on that channel the associated value is written to the end of the queue and the corresponding event flag is set. If the queue is already full, the last entry is overwritten. Only scalar items can be accommodated in the queue (if the channel is array-valued, only the first item will be written). The **pvGetQ** function reads items from the queue.

syncQ ***variable_name*** ***event_flag_name*** [***optional_queue_size***] **;**

Specifying Compiler Options

A compiler option is specified as follows:

option ***option_name*****;**

Possible options are given in Section 3. on page 15, and must include the "**+**" or "**–**" sign. Example:

```
option  +r;  /* make code reentrant */
```

Structure of a State Set

State_set is defined as:

ss ***state_set_name*** { ***state_def*** **. . .** }

State_def is defined as:

state ***state_name*** { [***optional_option_def***] **. . .** ***event_actions*** }

Optional_option_def is defined as:

option ***state_option_name*****;**

Event_actions is defined as:

[***optional_entry_action***] **. . .**
event_action **. . .**
[***optional_exit_action***] **. . .**

Optional_entry_action is defined as:

entry { ***statement*** **. . .** }

Event_action is defined as:

when (***expression***) { ***statement*** ... } state ***new_state***

Optional_exit_action is defined as:

exit { ***statement*** ... }

Any **`entry{}`** blocks are executed when the state is entered. **`exit{}`** blocks are executed when the state is left. See the options **`-e`** and **`-x`** below for more details about controlling this behavior. Note that the statements in all entry blocks of a state are executed before any of the expressions in **`when()`** conditions are evaluated.

Specifying State Options

Some options may be specified for a state using the **`option`** keyword. Currently there are three allowable options, **`t`**, **`e`** and **`x`**. The option string must be preceded by a "**+**" or "**–**", for instance **`option -te`**.

The options are:

`-t` Don't reset the time specifying when the state was entered if coming from the same state. When this option is used the delay() built-in function will return whether the given time delay 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 even if the previous state was the same as the current state.

`-x` Execute **`exit {}`** blocks even if the next state is the same as the current state.

`+t`, **`+e`** and **`+x`** are also permitted, though "**+**" is interpreted as "perform the default action for this option". For instance **`option +tx`** would have the same effect as if no option specification were given for **`t`** and **`x`**, so its use is only documentary. Note that more than one option line is allowed, and that syntax must be used to specify both "**+**" and "**–**" options, for instance:

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

Statements

A statement may be an assignment statement or an **`if`**, **`else`**, **`for`**, or **`while`** statement. These may contain expressions as follows:

- brackets: **`{ ... }`**
- variables (may have subscript)
- binary operators: **`+ - * / & | && << ...`**

- assignment operators: **= += *= ...**
- auto increment and auto decrement operators: **++ --**
- parenthesis
- pointer and address operators: *****
- structure operators: **. ->**
- functions

Although structure definitions and declarations are not recognized by the SNL, the structure operators are permitted.

Examples of statements in SNL:

```
pres3 = smooth (&p3[20], i+2);
for (j = 0; j < 10; j++)
{
   x[j] = 4.0*(y[j]/3.0 + sin(2.*pi*j));
}
```

Example of a state definition in SNL:

```
state low
{
   option -te;

   entry
   {
     printf("Will do this on each entry");
   }

   when(v>5.0)
   {
     printf("now changing to high\n");
   } state high

   when(delay(.1))
   {
     /* Pause of .1 on every iteration */
   } state low
}
```

Built-in Functions

The following special functions are built into the SNL. In most cases the state notation compiler performs some special interpretation of the parameters to these functions. Therefore, some are either not available through escaped C code or there use in escaped C code is subject to special rules. The term ***db_variable_name*** refers to any variable that is assigned to a database channel. When using such a variable, the function provides the association of the value or other characteristics of the channel to the variable.

delay

`int      delay(float` ***delay_in_seconds*** `)`

The delay function returns **`TRUE`** if the specified time has elapsed from entering the state. It should be used only within a **`when`** expression.

pvPut

`int   pvPut(`***db_variable_name***`)`

This function puts or writes the value to the database channel. The function returns the status from the channel access layer (e.g. **`ECA_NORMAL`** for success). It does not wait for the database channel write to be complete. Completion must be inferred by other means.

pvGet

`int   pvGet(`***db_variable_name***`)`

This function gets or reads the value from the database channel. The function returns the status from the channel access layer (e.g. **`ECA_NORMAL`** for success). By default, the state set will block until the read operation is complete. The asynchronous (**`+a`**) compile option should be used to prevent this.

pvGetQ

`int   pvGetQ(`***db_variable_name***`)`

This function removes the oldest value from a database channel's queue (the database channel should have been associated with a queue and an event flag via the **`syncQ`** statement) and updates the corresponding local sequencer variable. Despite its name, this function is really closer to **`efTestAndClear`** than it is to **`pvGet`**. It returns **`TRUE`** if the queue was not empty.

`pvGetQ` should only be called from within a **`when`** clause.

pvFreeQ

`int   pvFreeQ(`***db_variable_name***`)`

This function deletes all entries from a database channel's queue (the database channel should have been associated with a queue and an event flag via the **`syncQ`** statement).

pvGetComplete

`int   pvGetComplete(`***db_variable_name***`)`

This function returns **`TRUE`** if the last get for this channel is completed, i.e. the value in the variable is current. This call is appropriate only if the asynchronous (**`+a`**) compile option is specified.

pvMonitor

`int   pvMonitor(`***db_variable_name***`)`

This function initiates a monitor on the database channel.

pvStopMonitor

`int pvStopMonitor(`***db_variable_name***`)`

This function terminates a monitor on the database channel.

pvFlush

`int pvFlush()`

This function causes channel access to flush the input-output buffer. This call is appropriate only if the asynchronous (**`+a`**) compile option is specified.

pvCount

`int pvCount(`***db_variable_name***`)`

This function returns the element count associated with the database channel.

pvStatus

`int pvStatus(`***db_variable_name***`)`

This function returns the current alarm status for the database channel (e.g. **`HIHI_ALARM`**). The status and severity are only valid after a **`pvGet`** call or when a monitor returns.

Version 2.0: The value returned is one of the **`pvStat`** enumerations.

pvSeverity

`int pvSeverity(`***db_variable_name***`)`

This function returns the current alarm severity (e.g. **`MINOR_ALARM`**).

Version 2.0: The value returned is one of the **`pvSevr`** enumerations.

pvTimeStamp

`TS_STAMP pvTimeStamp(`***db_variable_name***`)`

This function returns the time stamp for the last **pvGet** or monitor of this variable. *The compiler does recognize type TS_STAMP. Therefore, variable declarations for this type should be in escaped C code. This will generate a compiler warning, which can be ignored.*

Version 2.0: The value returned is of type **`pvStamp`**.

pvAssign

`char* pvAssign(`***db_variable_name***`,` ***database_name***`)`

This function assigns or re-assigned the variable ***db_variable_name*** to ***database_name***. If ***database_name*** is an empty string or **`NULL`** then ***db_variable_name*** is de-assigned (not associated with any process variable).

pvAssigned

`int   pvAssigned(`***db_variable_name***`)`

This function returns **`TRUE`** if the channel is currently assigned.

pvConnected

`int   pvConnected(`***db_variable_name***`)`

This function returns **`TRUE`** if the channel is currently connected.

pvIndex

`int   pvIndex(`***db_variable_name***`)`

This function returns the channel index associated with a database channel. See "User Functions within the State Program" on page 32.

pvChannelCount

`int   pvChannelCount()`

This function returns the total number of channels associated with the state program.

pvAssignCount

`int   pvAssignCount()`

This function returns the total number of channels in this program that are assigned to database channels. Note: if all channels are assigned then the following expression is **`TRUE`**:

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

pvConnectCount

`int   pvConnectCount()`

This function returns the total number of channels in this program that are connected with database channels. Note: if all channels are connected then the following expression is **`TRUE`**:

```
pvConnectCount() == pvChannelCount()
```

efSet

`void  efSet(`***event_flag_name***`)`

This function sets the event flag and causes the execution of the **`when`** statements for all state sets that are pending on this event flag.

efTest

`int   efTest(`***event_flag_name***`)`

This function returns TRUE if the event flag was set.

efClear

`int efClear(`***event_flag_name***`)`

This function clears the event flag and causes the execution of the **`when`** statements for all state sets that are pending on this event flag.

efTestAndClear

`int efTestAndClear(`***event_flag_name***`)`

This function clears the event flag and returns **`TRUE`** if the event flag was set.

`efTestAndClear` should only be called from within a **`when`** clause.

macValueGet

`char* macValueGet(char *`***macro_name_string***`)`

This function returns a pointer to a string that is the value for the specified macro name. If the macro does not exist, it returns a **`NULL`**.

Comments

C-type comments may be placed anywhere in the program.

Escape to C Code

Because the SNL does not support the full C code standard, C code may be escaped in the program. The escaped code is not compiled by SNC, but is passed the “cc” compiler. There are two escape methods allowed:

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;
}%
```

If you are using the C preprocessor prior to compiling with **`snc`**, and you wish to defer interpretation of a preprocessor directive (**`#`** ***statement***), then you should use the form:

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

Any variable declared in escaped C code and used in SNL code will be flagged with a warning message by the SNC. However, it will be passed on to the C compiler correctly.

Exit Procedure

When a state set task is deleted, all state set tasks within the state program are also deleted. The state program may specify a procedure to run prior to task deletion. This is specified as follows:

exit { ***exit_code*** }

The exit code may be one or more statements as described above. However, no database functions may be called within the exit code.

This procedure should not be confused with the exit block of a state, which has the same syntax, but is executed at each transition from a state to the next state.

User Functions within the State Program

The last state set may be followed by C code, usually containing one or more user-supplied functions. Example:

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

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, which begin with the prefix **seq_** (e.g. **pvGet** becomes **seq_pvGet**). These C functions must pass a parameter identifying the calling state program, and if a database variable name is required, the *channel index* of that variable must be supplied. This channel index is obtained from the **pvIndex** function. Furthermore, if the code if complied with the **+r** option the database variables must be referenced as a structure element as described in "Variable Modification for Reentrant Option" on page 33. Examination of the intermediate C code that the compiler produces will indicate how to use the built-in functions and database variables.

Variable Extent

All variables declared in a state program are made static (non-global) in the C file, and thus are not accessible outside the state program module.

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 declarations in the SNL:

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

The C file will contain the following declaration:

```
struct UserVar {
   int     sw1;
   float   v5;
   short   wf2[1025];
};
```

The sequencer allocates the structure area at run time and passes a pointer to this structure into the state 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 the SNC for all SNL statements, but you will have to handle escaped C code yourself.

Default Run-time Parameters

Parameters to the state program may be supplied after the **`program`** statement within the SNL and as the second argument to the run-time sequencer. The format for parameters is:

"*macro_name* = *macro_value*, ..."

Examples:

```
program example ("logfile = example.log")
int     Vxy;
assign  Vxy to "HV{unit}:VXY";
```

At run-time the default for ***logfile*** can be over-ridden as follows:

```
seq &example, "logfile=ex1.log, unit=1"
```

The parameters specified at run time supersede those specified after the **`program`** statement. These parameters may also be used to specify the values for the macros used in the database names.

6. Examples of State Programs

Entry and exit action example

The following state 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");
     }
     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
   }
}
```

Dynamic assignment example

The following segment of a state 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 {
      ...
   }
}
```

Complex example

The following state 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 Section 2. on page 6. 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.

Preamble

```
/* File example.st: State program example. */
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;

char    *pmac;  /* used to access program macros */
```

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");
            /* Use global console fd */
            fd = ioGlobalStdGet(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 state 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
    }
}
```

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
   }
}
```

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
   }
}
```

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
   }
}
```

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;
}
```

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);
}
}%
```

New features in Version 1.9

New Language Features

With this version we have incorporated many extensions to the state notation language. Some of these changes offer significant advantage for programs and systems with a large number of database channels.

Number of Channels

The previous restriction on the number of database channels that could be defined no longer applies. Only the amount of memory on the target processor limits the number of channels.

Array Assignments

Individual elements of an array may be assigned to database channels. This feature simplifies many codes that contain groups of similar channels. Furthermore, double-subscripted arrays allow arrays of waveform channels.

Dynamic Assignments

Database channels may now be dynamically assigned or re-assigned within the language at run time.

Hex Constants

Hexadecimal numbers are now permitted within the language syntax. Previously, these had to be defined in escaped C code.

Time Stamp

The programmer now has access to the time stamp associated with a database channel.

Pointers

Variables may now be declared as pointers.

Sequencer Changes

The diagnostics included with the previous versions of the run-time sequencer were awkward to use and did not always provide relevant information. We corrected this shortcoming in this version.

seqShow

We enhanced the **`seqShow`** command to present more relevant information about the running state programs.

seqChanShow

The **`seqChanShow`** command now allows specification of a search string on the channel name, permits forward and backward stepping or skipping through the channel list, and optionally displays only channels that are not connected.

The syntax for displaying only channels that are not connected is
seqChanShow "<seq_program_name>","-"

ANSI Prototypes

SNC include files now use ANSI prototypes for all functions. To the programmer this means that an ANSI compiler must be used to compile the intermediate C code.

Fix for Task Deletion

Version 1.8 of the sequencer didn't handle the task deletion properly if a task tried to delete itself. We corrected this in version 1.9 (actually it was broken again at version 1.9.4 but was fixed at version 1.9.5).