DRAFT

State Notation Language and Run−time Sequencer Users Guide

Version 1.8

(for EPICS release 3.1 or later)

Manual Revision Date: 1 September, 1993

Written by Andy Kozubal
At−8

Not to be distributed outside Los Alamos National Laboratory without permission of the author.

This is an AT Division internal report and as such is a privileged communication. There is no warranty as to the accuracy of the contents. Should you wish to refer to the information contained herein, you should first ask the author. Acknowledgment should be made to the author, with the notation "Private communication."

Controls and Automation Group (AT−8)
Accelerator Technology Division
Mail Stop H820
Los Alamos National Laboratory
Los Alamos, NM 87544

1 Introduction

1.1 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−out 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 (AT−8), which is within the Accelerator Technology (AT) Division at the Los Alamos National Laboratory. The sequencer interfaces to the run−time database through the channel access facility of EPICS.

1.2 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 and the sequencer options.

1.3 Copyright and Restrictions

This software was produced under U.S. Government contract W−7405−ENG−36 at the Los Alamos National Laboratory, which is operated by the University of California for the U.S. Department of Energy. The EPICS software is copyright by the Regents of the University of California and the University of Chicago. This document may be reproduced and distributed without restrictions, provided it is reproduced in its entirety, including the cover page.

1.4 Other Documents

Listed below are other documents you may wish to consult.

Working in the EPICS Environment: Tutorial Slides
This document gives an overview of EPICS and describes how to get started. It explains much of the terminology and conventions (necessary reading if you are a first−time EPICS user).

DCT and Database Primer
This describes the key features of the EPICS run−time database, and describes how to configure and edit your own run−time database.

Channel Access Reference Manual
This describes how to access the run−time database through the channel access layer. This guide is not required unless you are programming in C or FORTRAN.

Display Editor User's Guide
This guide describes how to design and edit an operator interface display.

AR and ARR User Information Manual
This manual describes the EPICS archiver (AR) and archive retrieval (ARR) programs.

VxWorks Programmer's Guide
This manual describes the VxWorks development environment and the run−time interface to the VxWorks real−time operating system. A cursory knowledge of its contents will help you make better use of the sequencer.

VxWorks Reference Manual
This manual describes each VxWorks function or subroutine. You may want to refer to this manual for such things as file I/O and math routines.

Structured Development for Real−Time Systems
This three−volume set by Steve Mellor and Paul Ward (Yourdon Press) describes a methodology for modeling real−time systems using data flow diagrams and state transition diagrams.

2 State Notation Language Concepts

2.1 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.

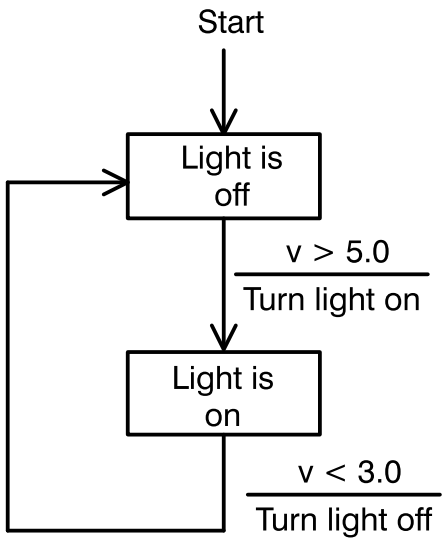


Figure 1: Example of a state transition diagram.

One way to express this diagram is: “From state ‘a’ when *event* occurs perform *action* then enter state ‘b’”. This is shown in figure 2, and provides a basis for the state notation language.

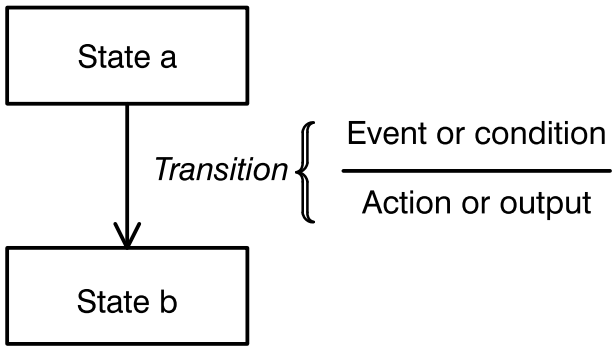


Figure 2: Elements of a state transition diagram.

2.2 Elements of the State Notation Language

The state transition diagram in figure 1 can be expressed by the following SNL:

```
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
}
```

Note 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 continually updated from the database. 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).

2.3 A Complete State Program

The complete state program for our example follows:

```
program level_check
float v;
assign v to ”ts1:ai1”;
monitor v;

short light;
assign      light to ”ts1:bo1”;

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
{
      . . . . . .
}
```

2.4 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 waveform with amplitude ¿11). This output is the same channel that is used as input by the “volt_check” state set.

```
. . . . . .
float vout;
float delta;
assign vout to ”ts1:ai1”;
. . . . . .
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, note that the “ramp” state always returns to itself. This is a permissible and often useful construction. The structure of this state set is show in the STD in figure 3.

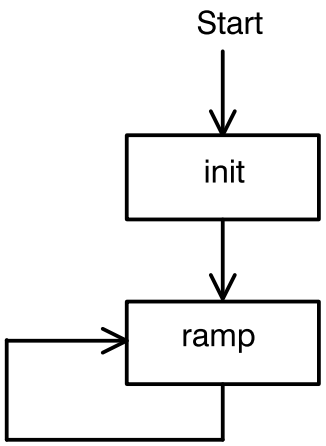


Figure 3: Structure of “generate_voltage” State Set

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 and 4. 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.

2.5 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*. A macro has a name (“unit” in this case). At run−time the macro is given 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 sections 3 and 4 for more on macros.

2.6 Data Types

The allowable variable declaration types correspond to the C types: “char”, “short”, “int”, “long”, “float”, and “double”. In addition there is the type “string”, which is a charater string of fixed length. 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 (see “Channel Access Users’ Manual). Strings are actually a fixed array size of type “char”.

You may specify variables as arrays. When assigned to a database channel the database operations are performed for the entire array

2.7 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 Appendix A, and the values for alarm status and severity are described in the DCT User’s Manual.

2.8 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 and event flag, and one or more state sets will test that event flag within a “when” clause. An event flag may also be associated with a monitor. 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. See Appendix B for an example using event flags.

2.9 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). The operation may not be initiated until the action statements in the current transition have been completed. and could complete at any later time. To test for completion use the function pvGetComplete()”, which is described in Appendix A.

2.10 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 (see section 3) 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 Appendix A. If a channel disconnects or re-connects during execution of a state program the sequencer updates the connection status appropriately.

2.11 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.

2.12 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.

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".

3.1 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:

```
snc test.st
cc -c test.c -I./epicsH -O -f68881
```

Alternatively, using the C pre−processor:

```
cc -E -o test.i test.st
snc test.i
cc -c test.c -I./epicsH -O -f68881
```

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). The above steps, which are summarized in figure 4, specify that the directory "./epicsH" be searched for include files. These specifications are necessary to pick up the various VxWorks and EPICS include files.

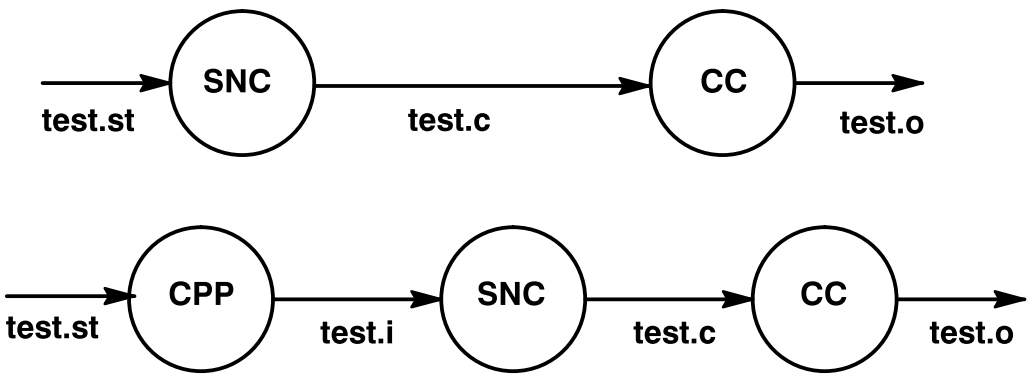


Figure 4: Two Methods of Compiling a State Program

3.2 Compiler Options

SNC provides four 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() and pvPut, i.e. the program will proceed before the operation is completed.

`-a` pvGet() and pvPut return 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.

`+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.

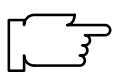
`+r` Make the run−time code reentrant, thus allowing more than one instance of the state program to run on an IOC.

`-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.

`+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).

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

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

3.3 Cross Compiler

When the target architecture is different from the host's, a cross compiler must be used. For instance, to produce MC68020 code on a Sun4 the command line might be:

```
cc -c test.c -I./h -I./vxh -O -f68881 -sun3
```

3.4 Compiling with a Makefile

A Makefile will help in compiling a state program. The document *Working in the EPICS Environment* describes the procedures for setting up a Makefile for EPICS Version 3.

3.5 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`" compiler option. However, this is not recommended unless you are familiar with the C file format and its relation to the SNL file.

3.6 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 flag.

3.7 Included Files

The following files are included with the ".c" file produced by SNC:

1. seq.h
2. cadef.h
3. db_access.h
4. fields.h
5. vxWorks.h
6. ioLib.h
7. semLib.h
8. ctype.h

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 the run−time sequencer so that you can execute your state program. We assume you are familiar with the VxWorks environment.

4.1 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, put into the "startup.cmd" file, or put into your own vxWorks command file.

4.2 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 you would use the following command:

```
seq &level_check
```

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

4.3 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:

```
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();
```

4.4 Specifying Run−Time Parameters

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"
```

Parameters can be accessed by your program with the function "macValueGet()", which is described in Appendix A.

The following built−in parameters have special meaning to the sequencer:

logfile: 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.

name: 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 = ***stack_size***
This parameter specifies the stack size in bytes. If the value is smaller than some minimum then this minimum stack size is used by the sequencer.

priority: 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.

4.5 Examining the State Program

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:

```
seqChanShow "level_check"
```

The parameter to "seqShow" and "seqChanShow" is either the task identifier (tid) or the task name of the state program task. If the the state program has more than one tid or name, then any one of these can be used. If you wish to see the task names and identifiers for all state programs type:

```
seqShow 0
```

4.6 Sequencer Logging

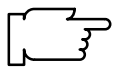
The sequencer logs various information that could help a user determine the health of a state program. For instance, during startup the database channels are listed as the program connects to them. Logging goes to the console by default, but may be directed to any file by specifying the *logfile* parameter as described above.

4.7 Implementation: What Triggers an Event?

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

1. a database value returns from database (monitor or pvGet)
2. an time delay has elapsed
3. an event flag is set
4. a channels connects and disconnects

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 *pvTestAndClear* or *pvClear*.



A State Notation Language Syntax

This section formalizes the state notation language syntax.

A.1 State Program

A state program has the following structure:

```
program program_name
        declarations
        state_set • • •
```

A.2 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 a channel access header file.

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];
```

A.3 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:

```
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.

The corresponding database request types are:

Variable Type	Monitor & pvGet	pvPut
char	DBR_STS_CHAR	DBR_CHAR
short	DBR_STS_SHORT	DBR_SHORT
int	DBR_STS_LONG	DBR_LONG
long	DBR_STS_LONG	DBR_LONG
float	DBR_STS_FLOAT	DBR_FLOAT
double	DBR_STS_DOUBLE	DBR_DOUBLE
string	DBR_STS_STRING	DBR_STRING

For 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.

A.4 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;
```

A.5 Declaring Event Flags

Event flags are declared as follows:

```
evflag               event_flag_name;
```

A.6 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;
```

A.7 Specifying Compiler Options

A compiler option is specified as follows:

```
option               option;
```

Possible options are given in section 3.2, and must include the "+" or "−" sign. Example:

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

A.8 Structure of a State Set

A ***state_set*** has the following structure:

```
ss state_set_name { state_def  • • • }
```

State_def is defined as:

```
state state_name { event_action • • • }
```

Event_action is defined as:

```
when (expression)
 { statement • • • } state new_state
```

A.9 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: +, -, *, /, &, |, &&, <<, etc.
assignment operators: =, +=, *=, etc.
auto increment and auto decrement operators
parenthesis
pointer and address operators: *,
structure operators: ., ->
functions
```

Although structure definitions and declartions 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));
}
```

A.10 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 “assign”ed 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(delay_in_seconds)
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).

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).

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.

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.

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.

pvSeverity

```
int pvSeverity(db_variable_name)
```

This function returns the current alarm severity (e.g. MINOR).

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 section A.14.

pvChannelCount

```
int pvChannelCount()
```

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

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()
```

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.

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)
```

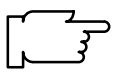
This function returns TRUE if the event flag was set.

efClear

```
int efClear(event_flag_name)
```

This function clears the event flag.

efTestAndClear



```
int efTestAndClear(event_flag_name)
```

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

macValueGet

```
char *macValueGet(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.

A.11 Notification of Monitor Completion

Event flags also allow notification of monitor completion. This is helpful where checking the monitored value is not practical.

```
sync variable_name   event_flag_name;

efTest(event_flag_name);
```

A.12 Comments

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

A.13 Escape to C Code

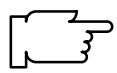
Because the SNL does not support the full C code standard, C code may be escaped in the program. The escaped code is not compiled by SNC, but is passed the the “cc” compiler. There are two escape methods allowed:

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.

A.14 Using Built−in Functions from C

Most of the above functions can be called from escaped C code. However, there are some conventions that must be followed. Most of these can be deduced by inspecting the C code generated by a state program. Within the SNL syntax the SNC automatically converts the database variable name to an index when it is used in one of the “pv” routines (e.g. pvPut). However, this substitution is not performed in escaped C code. The index may be obtained from the “pvIndex” macro as shown below:

```
int   index[20]
. . . .
            index[5] = pvIndex(pres_5);
. . . .
%%          pvPut(index[5]);
```

A.15 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 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.

A.16 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)
 • • •
}
}%
```

Normally, the built−in SNL functions such as pvGet() cannot be used in user−supplied functions.

A.17 Variable Extant

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.

A.18 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    *var_ptr;
```

Reference to variable “sw1” is made as:

```
var_ptr->sw1
```

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

A.19 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”)

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

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

B Example of a State Program

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. It generates a triangle waveform in one state set and detects the level in another. Other state sets detect and prints 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.

```
/*
 * 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;

%%static char *pmac;    /* used to access program macros */

/*====================== 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 connected\n");
          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 a ramp up/down */
ss generate_voltage {
     state init {
          when (efTest(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 (efTest(ef2)) {
          } state init
     }
}


/* Check for channel status; print exceptions */
ss test_status {
     state init {
          when (efTest(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
     }
}


/* Periodically write/read a waveform channel.  This uses
   pvGetComplete() to allow asynchronous pvGet(). */
ss periodic_read {
     state init {
          when (efTest(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 - close the log file */
exit {
     printf("close fd=%d\n", fd);
     if ((fd > 0) && (fd != ioGlobalStdGet(1)) )
          close(fd);
     fd = -1;
}

/*======================= 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(fd, value, status, severity)
int       fd;
float          value;
int       status, 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;
        }

        fdprintf(fd, ”Alarm condition: \”%s\””, pstr);

        if (severity == MINOR)
             pstr = ”minor”;
        else if (severity == MAJOR)
             pstr = ”major”;
        else
             pstr = ”none”;
        fdprintf(fd, ”, severity: \”%s\”, value=%g\n”, pstr, value);
}
}%

/*====================== End of state program
=====================*/
```

1 Introduction 1
1.1 Overview 1
1.2 Content of this Manual 1
1.3 Copyright and Restrictions 1
1.4 Other Documents 2

2 State Notation Language Concepts 3
2.1 The State Transition Diagram 3
2.2 Elements of the State Notation Language 4
2.3 A Complete State Program 5
2.4 Adding a Second State Set 6
2.5 Database Names − Using Macros 7
2.6 Data Types 7
2.7 Status of Database Channels 8
2.8 Synchronizing State Sets with Event Flags 8
2.9 Asynchronous Use of pvGet() 8
2.10 Connection Management 8
2.11 Multiple Instances and Reentrant Object Code 9
2.12 Database Variable Element Count 9

3 Compiling a State Program 10
3.1 The State Notation Compiler 10
3.2 Compiler Options 10
3.3 Cross Compiler 11
3.4 Compiling with a Makefile 11
3.5 Compiler Errors 11
3.6 Warnings from SNC 12
3.7 Included Files 12

4 Using the Run Time Sequencer 13
4.1 Loading a State Program 13
4.2 Executing the State Program 13
4.3 Deleting the State Program Tasks 13
4.4 Specifying Run−Time Parameters 13
4.5 Examining the State Program 14
4.6 Sequencer Logging 14
4.7 Implementation: What Triggers an Event? 14

A. State Notation Language Syntax 16
A.1 State Program 16
A.2 Declarations 16
A.3 Assignment of a Variable to a Database Channel 16
A.4 Monitoring a Database Channel 17
A.5 Declaring Event Flags 17
A.6 Associating an Event Flag with a Database Channel 17
A.7 Specifying Compiler Options 17
A.8 Structure of a State Set 17
A.9 Statements 18
A.10 Built−in Functions 18
A.11 Notification of Monitor Completion 20
A.12 Comments 21
A.13 Escape to C Code 21

A.14 Using Built−in Functions from C 21
A.15 Exit Procedure 22
A.16 User Functions within the State Program 22
A.17 Variable Extant 22
A.18 Variable Modification for Reentrant Option 22
A.19 Default Run−time Parameters 23
B Example of a State Program 24