State Notation Language and Sequencer Users' Guide

William Lupton (wlupton@keck.hawaii.edu)
W. M. Keck Observatory
Kamuela, HI 96743, USA
March 2000
EPICS Release 3.14.0

Table of Contents

Table of Contents iii

Chapter 1: Introduction 1

1.1. Note on Versions 1
1.2. Overview 1
1.3. Content of this Manual 2
1.4. Copyright and Restrictions 2
1.5. Notes on This Release 2
1.6. Future Plans 5
1.7. Notes on v1.9 Release 5

Chapter 2: State Notation Language Concepts 7

2.1. The State Transition Diagram 7
2.2. Elements of the State Notation Language 8
2.3. A Complete State Program 8
2.4. Adding a Second State Set 9
2.5. Variable Names Using Macros 10
2.6. Data Types 11
2.7. Arrays of Variables 11
2.8. Dynamic Assignment 12
2.9. Status of Control System Variables 12
2.10.Synchronizing State Sets with Event Flags 13
2.11.Queuing Monitors 14
2.12.Asynchronous Use of pvGet() 14
2.13.Asynchronous Use of pvPut() 15
2.14.Connection Management 16
2.15.Multiple Instances and Reentrant Object Code 16
2.16.Control System Variable Element Count 16

Chapter 3: Compiling a State Program 19

3.1. Files needed . . . 19
3.2. The State Notation Compiler . . . 19
3.3. Name of output file . . . 20
3.4. Compiler Options . . . 20
3.5. Compiler Errors . . . 21
3.6. Compiler Warnings . . . 21
3.7. Compiling and Linking a State Program under Unix . . . 21
3.8. Using makeBaseApp . . . 22

Chapter 4: Using the Run Time Sequencer . . . 25
4.1. VxWorks-specific instructions . . . 25
4.2. Unix-specific instructions . . . 26
4.3. Specifying Run-Time Parameters . . . 27
4.4. Sequencer Logging . . . 28
4.5. What Triggers an Event? . . . 28

Chapter 5: State Notation Language Syntax . . . 29
5.1. Typographical conventions . . . 29
5.2. State Program . . . 29
5.3. Definitions . . . 30
5.4. State Sets . . . 32
5.5. Statements and Expressions . . . 34
5.6. Built-in Functions . . . 35
5.7. C Compatibility Features . . . 38

Chapter 6: The PV (Process Variable) API . . . 41
6.1. Introduction . . . 41
6.2. Rationale . . . 41
6.3. A tour of the API . . . 42
6.4. The API in More Detail . . . 43
6.5. Supporting a New Message System . . . 48

Chapter 7: Examples of State Programs . . . 53
7.1. Entry and exit action example . . . 53
7.2. Dynamic assignment example . . . 54
7.3. Complex example . . . 54

Chapter 8: Installation . . . 59
8.1. Prerequisites . . . 59
8.2. Obtaining the distribution . . . 59
8.3. Unpacking the distribution . . . 60
8.4. Preparing to build . . . 60
8.5. Building and installing . . . 61
8.6. Verifying the installation . . . 61
8.7. Using the installation . . . 63

Chapter 9: Acronyms/Glossary . . . 65
9.1. Acronym List . . . 65
9.2. Glossary . . . 65

Index . . . 67

Chapter 1: Introduction

1.1 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 version 2.0, for which the changes have been implemented by William Lupton of W. M. Keck Observatory and Greg White of Stanford Linear Accelerator Center (SLAC).

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

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

1.2 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 control system variables 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, free electron lasers and telescopes. 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 underlying control system through a generic PV (process variable) API that supports, among other message systems, the channel access facility of EPICS.

1.3 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. Then, we present a complete description of the SNL syntax and the sequencer options. Finally, we describe the PV layer, give some examples of working sequences, and describe how to build, test and work with the sequencer installation.

1.4 Copyright and Restrictions

This software was produced under U.S. Government contract at Los Alamos National Laboratory and at Argonne National Laboratory. 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.

1.5 Notes on This Release

New version 1.9 features have been moved to Section 1.7 on page 5. This section gives brief notes on version 2.0 changes.

Version 2.0 of the sequencer and state notation compiler is available for EPICS release R3.14 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, under most circumstances no source-level changes to existing programs are required.

1.5.1 Portability changes

These changes allow state programs to run unchanged on hosts and IOCs.

Replaced VxWorks dependencies with OSI routines

All VxWorks routines have been replaced with the appropriate OSI (Operating System Independent) routines. State programs can run in any environment for which there is an OSI implementation.

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. See Chapter 6: on page 41.

Added optional generation of main program

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

When this option is enabled, the main thread reads from standard input and can execute **seqShow**, **seqChanShow** *etc.* on demand. End of file causes the sequencer to exit.

1.5.2 New Language Features

Entry handler

A one-off **entry** handler can be supplied (*c.f.* the existing **exit** handler). This is called once, at sequencer start-up, in the context of the first state set, before the remaining state set threads have been created. See “entry_handler” on page 30.

Entry and exit actions

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. Note that these blocks are aassociated with a state and are not the same as the one-off entry and exit handlers. See “entry_action” on page 33 and “exit_action” on page 33.

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 complementary to **-e**, but for the new **exit** block. See “state_option_stmt” on page 33.

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. When SNL arrays are used, a single queue is shared by the control system variables associated with the elements of the array, which can be useful for parallel control. See “Queuing Monitors” on page 14 and “syncq_stmt” on page 32.

Device support

An device support module has been added. This allows EPICS records to refererence sequencer internals. At present this is very basic and can only return state-set names. See Section 8.6 on page 61 for a well-hidden example (look for “**caget ss0**”).

Local variables

SNL does not support the declaration of local variables. However, the C code generated for a `when` clause is now placed within an extra level of braces and the C escape mechanism can be used to declare a local variable. See “Variable Extent” on page 39.

More functions are safe in action code

In previous versions, some functions, *e.g.* `pvPut`, have acquired a resource lock and others, *e.g.* `efTestAndClear`, have not. Those that didn’t were intended for use in action code and those that did not were intended for use in `when` clauses. This was confusing and dangerous. All such functions now acquire a mutex (that can be taken recursively).

Asynchronous puts

`pvPut` can now put process variables asynchronously by using an extra `ASYNC` argument. Completion can be tested using the new `pvPutComplete`. Arrays are supported (so `pvPutComplete` can be used to test whether a set of puts has completed). See Section 2.13 on page 15 and “pvPutComplete” on page 35.

Synchronous/asynchronous override on gets and puts

`pvGet` and `pvPut` both accept an optional `SYNC` or `ASYNC` argument that, for `pvGet`, overrides the default as set using the `-a` option and, for `pvPut`, overrides the default synchronous behavior. See “pvPut” on page 35 and “pvGet” on page 36.

Sequencer deletion re-written

Sequencer deletion has been completely re-written. You can no longer delete a sequencer by deleting one of its tasks. Instead you must use the new `seqStop` routine. See “Stopping the State Program Tasks” on page 26.

efClear can wake up state sets

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

More C syntax is supported

The “`to`” in `assign`, `sync` and `syncQ` statements is now optional.

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 (relaxes restrictions on using `#include` to include code).

“`~`” (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).

1.5.3 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

SEGV no longer occurs when doing `seqShow` and there was no previous state

Miscellaneous other problems found by purify were fixed.

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.

1.5.4 Miscellaneous

Compilation warnings have been avoided wherever possible.

A 60Hz system clock frequency is no longer assumed.

Error reporting is now more consistent; it is currently just using **`errlogPrintf`**.

The new EPICS R3.14 **`configure`**-based make rules are used.

1.6 Future Plans

Several items remain unsupported or only partially supported. Users are encouraged to provide feedback on this list or on other desired items.

Device support

This is partially supported. See “Device support” on page 3.

Local variables

These are partially supported. See “Local variables” on page 4.

pvNew dynamic loading

This would remove some undesirable library dependencies. See “Overview” on page 42.

Hierarchical states

This would be a major enhancement and would, incidentally, bring the sequencer model into very close agreement with the Harel model that is espoused by the UML. Events would be propagated up the state hierarchy.

1.7 Notes on v1.9 Release

With this version (v1.9), we have incorporated many extensions to the state notation language. Some of these changes offer significant advantages for programs and systems with a large number of control system variables.

Number of control system variables

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

Array assignments	Individual elements of an array may be assigned to control system variables. This feature simplifies many codes that contain groups of similar variables. Furthermore, double-subscripted arrays allow arrays of array-valued variables.
Dynamic assignments	Control system variables 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 control system variable.
Pointers	Variables may now be declared as pointers.
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 variable name, permits forward and backward stepping or skipping through the variable list, and optionally displays only variables that are or are not connected. The syntax for displaying only variables 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.

Chapter 2: State Notation Language Concepts

2.1 The State Transition Diagram

The state transition diagram or STD is a graphical notation for specifying the behavior of a control system in terms of control transformations. The 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 on the current memory or state. For instance, specifying an input of 4.2 volts does not directly specify the output; that depends on the current state.

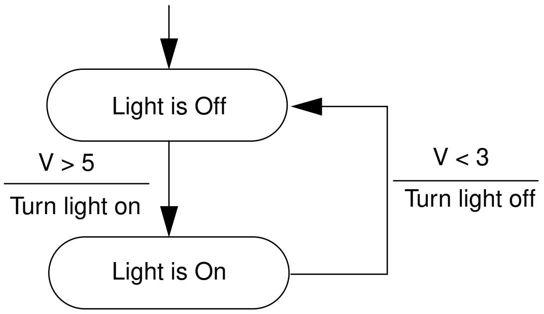

Figure 2-1: A simple state transition diagram

2.2 Elements of the State Notation Language

The following SNL code segment expresses the STD in Figure 2-1 on page 7:

```
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 control system variables. 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 control system variables 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 control system variables "**`Input_voltage`**" and "**`Indicator_light`**" respectively. We want the value of **`v`** to be updated automatically whenever it changes. This is accomplished with the following declaration:

```
monitor   v;
```

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

2.3 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 { ... }
```

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 function with amplitude 11).

First, we add the following lines to the declaration:

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

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

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

```
    }
}
```

The above example exhibits several concepts. First, note that the **`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 construct. The structure of this state set is shown in the STD in Figure 2-2 on page 10.

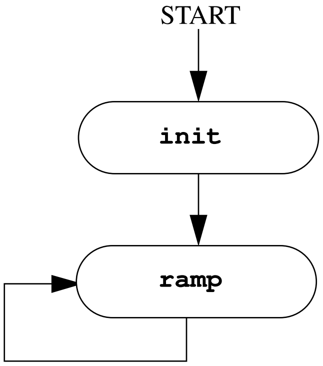

Figure 2-2: Structure of **`generate_voltage`** State Set

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 Chapters 3-5. You probably want to pick unique names for your control system variables, 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 Variable Names Using Macros

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

```
assign v 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 control system variable name. For example, if the macro "**`unit`**" is given a name "**`DTL_6:CM_2`**", then the run-time variable 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 4.3 on page 27 for more on macros.

2.6 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`** (at the time of writing, a string can hold 40 characters). Sequencer variables having any of these types may be assigned to a control system variable. The type declared does not have to be the same as the native control system value type. The conversion between types is performed at run-time.

You may specify array variables as follows:

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

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

2.7 Arrays of Variables

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

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

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

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

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

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

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

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

Alternatively, we can do this for the entire array:

```
monitor   Vin;
```

And the same goes when synchronizing with event flags (Section 2.10 on page 13) and queuing monitors (Section 2.11 on page 14). SNL arrays are really quite powerful.

Double subscripts offer additional options.

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

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

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

2.8 Dynamic Assignment

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

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

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()
```

Having assigned a variable, you should wait for it to connect before using it (although it is OK to monitor it). See Section 2.14 on page 16.

2.9 Status of Control System Variables

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

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

```
...
```

These routines are described in Section 5.1 on page 29. The values for status and severity are defined in the include file **`pvAlarm.h`**, and the time stamp is returned as a standard EPICS **`TS_STAMP`** structure, which is defined in **`tsStamp.h`**. Both these files are automatically included when compiling sequences (but the SNL compiler doesn't know about them, so you will get warnings when using constants like **`pvStatOK`** or tags like **`secPastEpoch`**).

2.10 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 control system variable that is being monitored. In that case, whenever a monitor is delivered, the corresponding event flag is set. Note that this provides an alternative to testing the value of the monitored channel and is particularly valuable when the channel being tested is an array or when it can have multiple values and an action must occur for any change.

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

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

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

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

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

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

2.11 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 variable may transition to 1 and then back to 0 to indicate that a command is active and has completed. These transitions may occur in rapid succession. This problem can be avoided by using the **`syncQ`** statement to associate a variable with both a queue and an event flag. The **`pvGetQ`** function retrieves and removes the head of queue.

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

```
long command; assign command to “commandVar”;

long active; assign active to “activeVar”; monitor active;
evflag activeFlag; syncQ active activeFlag;

    state start {
        when () {
            pvFreeQ( active );
            command = 1;
            pvPut( command );
        } state sent
    }

    state sent {
        when ( pvGetQ( active ) && active ) {
        } state high
    }

    state high {
        when ( pvGetQ( active ) && !active ) {
        } state done
    }
```

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

2.12 Asynchronous Use of pvGet()

Normally the **`pvGet`** operation completes before the function returns, thus ensuring data integrity. However, it is possible to use these functions asynchronously by specifying the **`+a`** compiler flag (see Section 3.4 on page 20). The operation might not be initiated until the action statements in the current transition have been completed and it could complete at any later time. To test for completion use the function **`pvGetComplete`**, which is described in “pvMonitor” on page 36.

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

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

2.13 Asynchronous Use of pvPut()

Normally the **`pvPut`** operation completes asynchronously. In the past it has been the responsibility of the programmer to ensure that the operation completed (typically by monitoring other variables). However, the function **`pvPutComplete`** can now be used for this. Also, while the **`+a`** compiler flag does not affect put operations, **`pvPut`**, like **`pvGet`**, accepts an optional **`SYNC`** or **`ASYNC`** argument, which forces a synchronous or asynchronous put. For example:

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

`pvPutComplete` supports arrays and can be used to check whether a set of puts have all completed. This example illustrates how to manage a set of parallel commands.

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

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

    state active {
        when ( pvPutComplete( init ) ) {
        } state done

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

`pvPutComplete` also supports optional arguments to wake up the state set as each put completes. The following could be inserted before the first **`when`** clause in the **`active`** state above. The **`TRUE`** argument causes **`pvPutComplete`** to return **`TRUE`** when any command completed (rather than only when all commands complete). The **`done`** argument is the address of a **`long`** array of the same size as **`init`**; its elements are set to 0 for puts that are not yet complete and to 1 for puts that are complete.

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

2.14 Connection Management

All control system variable connections are handled by the sequencer via the PV API. Normally the state programs are not run until all control system variables are connected. However, with the **`-c`** compiler flag, execution begins while the connections are being established. The program can test for each variable's connection status with the **`pvConnected`** routine, or it can test for all variables connected with the following comparison (if not using dynamic assignment, Section 2.8 on page 12, **`pvAssignCount`** will be the same as **`pvChannelCount`**):

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

These routines are described in Section 5.6 on page 35. If a variable disconnects or re-connects during execution of a state program, the sequencer updates the connection status appropriately; this can be tested in a **`when`** clause, as in:

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

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

```
when (pvConnectCount() == pvAssignCount()) {
} state connected
when (delay(10)) {
} state connect_timeout
```

Note that the connection callback may be delivered before or after the initial monitor callback (the PV API does not specify the behavior, although the underlying message system may do so). If this matters to you, you should synchronize the value with an event flag and wait for the event flag to be set before proceeding. See Section 2.10 on page 13 for an example.

2.15 Multiple Instances and Reentrant Object Code

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

2.16 Control System Variable Element Count

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

```
float     wf[2000];
assign    wf to "{unit}:CavField.FVAL";
int       LthWF;
```

```
...
        LthWF = pvCount(wf);
        for (i = 0; i < LthWF; i++) {
            ...
        }
        pvPut(wf);
        ...
```

Chapter 3: Compiling a State Program

The sequencer is distributed as an EPICS R3.14 **`makeBaseApp`** application. The first sections of this chapter show how to build a Unix sequence independent of any particular build environment. These sections are followed by a section describing how to use **`makeBaseApp`** to build sequences. See Chapter 3: on page 19 for installation instructions.

3.1 Files needed

In order to compile and run an EPICS sequence, a C/C++ compiler and the following files are required.

1. **`snc`**, the SNL compiler
2. sequencer include files **`seqCom.h`** and **`pvAlarm.h`**
3. EPICS include files **`shareLib.h`**, **`epicsTypes.h`** and **`tsStamp.h`**
4. if using the **`+m`** compiler option, EPICS include files **`osiThread.h`**, **`osiSem.h`**, **`errlog.h`** and **`taskwd.h`** (and files included by them)
5. sequencer libraries **`libseq`**, **`libpv`** and **`libpvXxx`** (for message systems, *e.g.* **`libpvCa`** for CA); on an IOC, these are linked into **`seqLibrary`** and **`pvLibrary`**
6. libraries for any message systems other than CA
7. EPICS libraries **`libca`** (if using CA) and **`libCom`**; on an IOC, these are linked into **`iocCoreLibrary`**

3.2 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 may be used prior to 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 test.o ...additional compiler options
gcc test.o -o test ...additional loader options
```

Alternatively, using the C pre-processor:

```
gcc -E -x c test.st >test.i
snc test.i
gcc -c test.c -o test.o ...additional compiler options
gcc test.o -o test ...additional loader options
```

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`**).

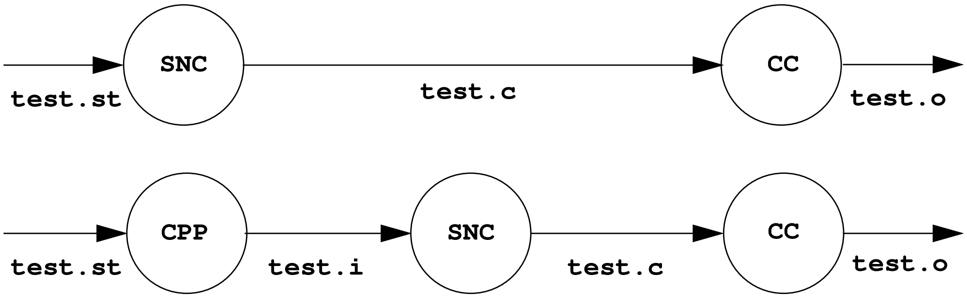

Figure 3-1: Two Methods of Compiling a State Program

3.3 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, the **`-o`** compiler option overrides.

3.4 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;
```

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** (“ell”) 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 Compiler Warnings

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.

3.7 Compiling and Linking a State Program under Unix

Under Unix, either the **+m** compiler option should be used to create a C main program or else the programmer should write a main program (the main program plays the same role as the VxWorks startup script).

Here is a full build of a simple state program from source under Solaris. Compiler and loader options will vary with other operating systems. It is assumed that the sequencer is in **/usr/local/epics/seq** and that EPICS is in **/usr/local/epics**.

```
gcc -E -x c demo.st >demo.i

snc +m demo.i

gcc -D_POSIX_C_SOURCE=199506L -D_POSIX_THREADS -D_REENTRANT \
    -D__EXTENSIONS__ -DnoExceptionsFromCXX \
    -DOSITHREAD_USE_DEFAULT_STACK \
    -I. -I.. \
    -I/usr/local/epics/seq/include \
    -I/usr/local/epics/base/include/os/solaris \
    -I/usr/local/epics/base/include -c demo.c

g++ -o demo \
    -L/usr/local/epics/seq/lib/solaris-sparc \
    -L/usr/local/epics/base/lib/solaris-sparc \
    demo.o -lseq -lpv -lpvCa -lca -lCom \
    -lposix4 -lpthread -lthread -lsocket -lnsl -lm
```

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

```
/* Main program */
#include "osiThread.h"
#include "errlog.h"
#include "taskwd.h"

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

The arguments are essentially the same as those taken by the **seq** routine.

3.8 Using makeBaseApp

The sequencer is distributed as an EPICS R3.14 **makeBaseApp** application. This section doesn't describe how to install and build the sequencer itself (for that, refer to Chapter 8: on page 59); instead, it describes how to build a sequencer application.

Makefile

Assume a sequencer in **demo.st**. This sequencer will use the CA message system. It is to be linked into a Unix program called **demo** and a VxWorks object file called **demo.o**. Also assume that the sequencer includes and libraries can be accessed via **SEQ** (and that **SEQ_LIB** is defined; see Section 8.7 on page 63). The following **Makefile** does the job.

```
TOP = ../..
include $(TOP)/configure/CONFIG

SNCFLAGS = +m

SEQS = demo
PROD = $(SEQS)
OBJS_vxWorks = $(SEQS)
```

```
PROD_LIBS += seq pv pvCa ca Com
seq_DIR = $(SEQ_LIB)

include $(TOP)/configure/RULES
```

Make output

When I build with the above **`Makefile`** on my Solaris machine with a Power PC IOC I get the following output:

```
% gmake

gnumake -C O.solaris-sparc -f ../Makefile TOP=../../.. \
   T_A=solaris-sparc install
preprocessing demo.st
/usr/local/bin/gcc -x c -E  -D_POSIX_C_SOURCE=199506L \
   -D_POSIX_THREADS -D_REENTRANT -D__EXTENSIONS__ \
   -DnoExceptionsFromCXX -DOSITHREAD_USE_DEFAULT_STACK \
   -DUNIX -I. -I/home/wlupton/epics/seq/include \
   -I../../../include/os/solaris -I../../../include \
   -I/home/wlupton/epics/anl/base/include/os/solaris \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/seq/include -I.. ../demo.st > demo.i
converting demo.i
/home/wlupton/epics/seq/bin/solaris-sparc/snc +m demo.i
/usr/local/bin/gcc -ansi -pedantic  -D_POSIX_C_SOURCE=199506L \
   -D_POSIX_THREADS -D_REENTRANT -D__EXTENSIONS__ \
   -DnoExceptionsFromCXX -DOSITHREAD_USE_DEFAULT_STACK \
   -DUNIX  -O3 -g -g -Wall -I. -I/home/wlupton/epics/seq/include \
   -I../../../include/os/solaris -I../../../include  \
   -I/home/wlupton/epics/anl/base/include/os/solaris \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/seq/include -I.. -c demo.c
/usr/local/bin/g++ -ansi -pedantic -Wtraditional -o demo \
   -L/home/wlupton/epics/anl/base/lib/solaris-sparc/ \
   -L/home/wlupton/epics/seq/lib/solaris-sparc/ \
   -R/home/wlupton/epics/anl/base/lib/solaris-sparc/ \
   -R/home/wlupton/epics/seq/lib/solaris-sparc/ demo.o \
   -lseq -lpv -lpvCa -lca -lCom -lposix4 -lpthread -lthread \
   -lsocket -lnsl  -lm
Installing binary ../../../bin/solaris-sparc/demo

gnumake -C O.vxWorks-ppc604 -f ../Makefile TOP=../../.. \
   T_A=vxWorks-ppc604 install
preprocessing demo.st
GCC_EXEC_PREFIX=/usr/local/vw/t2/host/sun4-solaris2/lib/gcc-lib/ \
   /usr/local/vw/t2/host/sun4-solaris2/bin/ccppc -nostdinc -x c \
   -E -nostdinc -DnoExceptionsFromCXX -DCPU=PPC604 -DvxWorks -I. \
   -I/home/wlupton/epics/seq/include \
   -I../../../include/os/vxWorks -I../../../include  \
   -I/home/wlupton/epics/anl/base/include/os/vxWorks \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/seq/include -I.. \
   -I/usr/local/vw/t2/target/h ../demo.st > demo.i
converting demo.i
```

```
/home/wlupton/epics/seq/bin/solaris-sparc/snc +m demo.i
GCC_EXEC_PREFIX=/usr/local/vw/t2/host/sun4-solaris2/lib/gcc-lib/ \
   /usr/local/vw/t2/host/sun4-solaris2/bin/ccppc -nostdinc -ansi \
   -pedantic -B/usr/local/vw/t2/host/sun4-solaris2/lib/gcc-lib/ \
   -nostdinc -DnoExceptionsFromCXX -DCPU=PPC604 -DvxWorks -O2 -g \
   -Wall -mcpu=604 -mlongcall -fno-builtin -I. \
   -I/home/wlupton/epics/seq/include \
   -I../../../include/os/vxWorks -I../../../include \
   -I/home/wlupton/epics/anl/base/include/os/vxWorks \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/anl/base/include \
   -I/home/wlupton/epics/seq/include -I.. \
   -I/usr/local/vw/t2/target/h -c demo.c
Installing binary ../../../bin/vxWorks-ppc604/demo.o
```

Chapter 4: Using the Run Time Sequencer

In the previous chapter you learned how to create and compile some simple state programs. In this chapter you will be introduced to the run-time sequencer so that you can execute your state program.

4.1 VxWorks-specific instructions

Note that the latest sequencer version has not, at the time of writing, been fully checked out under VxWorks.

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 < pvLibrary
ld < seqLibrary
```

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.

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 control system variables 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 *unquoted* 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 ***<Enter>*** 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
```

Stopping the State Program Tasks

You can no longer directly delete state program tasks. Instead, you must use **`seqStop`**.

```
seqStop level_check
```

The parameter to **`seqStop`** is either the task identifier (tid) or the *unquoted* task name of the state program task.

A state program can no longer delete itself.

4.2 Unix-specific instructions

Executing the State Program

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

```
level_check
```

Once the state set threads have been created, the console remains active and you can type commands as described below.

Examining the state program

The following commands can be issued under Unix (hit "**?**" to obtain the list):

```
commands (abbreviable):
  i         - show all threads
```

```
  all       - show all sequencers
  channels  - show all channels
  +         - show conn. channels
  -         - show disc. channels
  queues    - show queues
  statesets - show state-sets
  <EOF>     - exit
```

As you see, all commands can be abbreviated to a single character.

Stopping the State Program Tasks

A state program may be killed by sending it a **`SIGTERM`** (***Ctrl-C***) signal (this is an untidy exit, but who cares?) or by entering an ***<EOF>*** (***Ctrl-D***) character. The latter calls **`seqStop`** and is a tidy exit.

4.3 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, ..."

This same format can be used in the **`program`** statement's parameter list (Section 5.2 on page 29). Parameters specified on the command-line override those specified in the **`program`** statement.

VxWorks

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"
```

Unix

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

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

Access within program

Parameters can be accessed by your program with the function **`macValueGet`**, which is described in Section on page 38. The following built-in parameters have special meaning to the sequencer:

`debug =` ***level***

Sets a logging level. ***level-1*** is passed on to the PV API. Can be used in user code.

`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 **`stdout`**.

`name =` ***thread_name***

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

`priority =` ***task_priority***

This parameter specifies the initial task priority when the tasks are created. The value ***task_priority*** must be an integer between 0 and 99 (it's ignored under Unix).

`stack =` ***stack_size***

This parameter specifies the stack size in bytes (its use is deprecated, and it is in any case ignored under Unix).

4.4 Sequencer Logging

The sequencer logs various information that could help a user determine the health of a state program. Logging uses the **`errlogPrintf`** function and will be directed to the IOC log file if the IOC log facility has been initialized. Under VxWorks this is done automatically but under Unix it must be done by the programmer. This can be done in the main program (if you are writing it yourself) or in the entry handler, which is executed in the context of the first state set before the remaining state sets have been created. For example:

```
entry {
#ifdef UNIX
%%#include "logClient.h"
    iocLogInit();
#endif
}
```

The programmer may log information using **`errlogPrintf`** directly or else by using the **`seqLog`** function. By default, **`seqLog`** output goes to **`stdout`**, but it may be directed to any file by specifying the **`logfile`** parameter as described above.

4.5 What Triggers an Event?

There are five types of sequencer event:

- a control system variable monitor is posted
- an asynchronous **`pvGet`** or **`pvPut`** completes
- a time delay elapses
- an event flag is set or cleared
- a control system variable connects or 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, event flags were cleared after a **`when`** statement executed. Currently, event flags must be cleared with either **`efTestAndClear`** or **`efClear`**, unless the **`-e`** compiler option was chosen.

Chapter 5: State Notation Language Syntax

This chapter formalizes the state notation language syntax using a variant of BNF (Backus-Naur Form).

5.1 Typographical conventions

The idea is that the meaning will be clear without explanation. However, here are some explanatory notes.

- words and symbols 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 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

5.2 State Program

program

`program` ***program_name*** **[**`("`***parameter_list***`")`**]**`;`
[entry_handler]
definition...
state_set...
[exit_handler]

program_name

The name of the program. This is used as the name of the global variable which contains or points to all the state program data structures (the address of this global variable is passed to the **`seq`** function when creating the run-time sequencer). It is also used as the base for the state set thread names unless overridden via the **`name`** parameter (Section 4.3 on page 27).

parameter_list

A list of comma-separated parameters in the same form as they are specified on the command line (Section 4.3 on page 27). Command-line parameters override those specified here.

definition

See Section 5.3 on page 30.

entry_handler

A state program may specify entry code to run prior to state set thread creation. This is run in the context of the first state set thread, before the other threads are created and is specified as follows:

```
entry {
    [statement]...;
}
```

The entry code consists of zero or more statements as described in Section 5.5 on page 34. However, no control system variable access functions may be called within the entry code.

This handler should not be confused with the entry block of a state, which has the same syntax, but is executed at each transition to a new state.

state_set

See Section 5.4 on page 32.

exit_handler

When a state program is stopped via **`seqStop`**, all state set threads within the state program are deleted. The state program may specify exit code to run prior to thread deletion. This is run in the context of the first state set thread and is specified as follows:

```
exit {
    [statement]...;
}
```

The exit code consists of zero or more statements as described in Section 5.5 on page 34. However, no control system variable access functions may be called within the exit code.

This handler 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.

5.3 Definitions

definition

definition = ***decl_stmt*** | ***assign_stmt*** | ***monitor_stmt*** | ***sync_stmt*** | ***syncq_stmt*** | ***compiler_option_stmt***

decl_stmt

Variable declarations are similar to C except that the types are limited to the following, only scalar 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***`;`
`evflag` ***event_flag_name***`;`

Type **`string`** produces an array of char with length equal to the constant **`MAX_STRING_SIZE`**, which is defined (as 40) in one of the included header files. Unsigned types and pointer types may also be specified. For example:

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

Variables 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 arrays of strings and event flags are not implemented.

assign_stmt

Once a variable is declared, it may be assigned to a control system variable. Thereafter, that variable is used to interact with the underlying control system. The following are all variations on assignment (note that the "**`to`**" is optional):

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

A control system variable name may contain one or more macro names enclosed in braces, as in "**`{sys}{sub}voltage`**". Macros are named following the same rules as C language variables.

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

monitor_stmt

To make the state program event-driven, input variables can be monitored. Monitored variables are automatically updated with the current value of the underlying control system variable (the variable must first be assigned to a control system variable).

`monitor` ***variable_name***`;`
`monitor` ***variable_name***`[`***index***`];`

sync_stmt

An event flag can be associated with an SNL variable (which may be an array, and thus associated with several control system variables). When a monitor is posted on any of the associated control system variables, the corresponding event flag is set (even if it was already set). Note that the “**`to`**” is optional.

`sync` ***variable_name*** **`[`**`to`**`]`** ***event_flag_name*** `;`
`sync` ***variable_name***`[`***index***`]` **`[`**`to`**`]`** ***event_flag_name*** `;`

syncq_stmt

An event flag can be associated with a monitor queue which, in turn, is associated with an SNL variable (which may be an array, and thus associated with several control system variables). The queue size defaults to 100 but can be overridden on a per-queue basis. When a monitor is posted on any of the associated control system variables, the variable’s 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 variable is array-valued, only the first item will be saved). The **`pvGetQ`** function reads items from the queue.

`syncQ` ***variable_name*** **`[`**`to`**`]`** ***event_flag_name [queue_size]*** `;`
`syncQ` ***variable_name***`[`***index***`]` **`[`**`to`**`]`** ***event_flag_name [queue_size]*** `;`

Note that the square brackets around “**`to`**” and ***queue_size*** indicate optional items rather than literal square brackets.

compiler_option_stmt

A compiler option is specified as follows:

`option` ***compiler_option_name*** `;`

Possible compiler options are given in Section 3.4 on page 20, and must include the “**+**” or “**–**” sign. Example:

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

5.4 State Sets

state_set

```
ss state_set_name {
    state_def...
}
```

state_set_name

The name of the state set. The normal C variable naming rules apply.

state_def

```
state state_name {
    [state_option_stmt]...
    [entry_action]...
    event_action...
    [exit_action]...
}
```

state_name

The name of the state. The normal C variable naming rules apply. State names need only be unique within the state set (*e.g.* each state set within a state program could have a **`start`** state).

state_option_stmt

A state option is specified as follows:

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

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`** 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 example:

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

entry_action

```
entry {
    [statement]...
}
```

`entry` blocks are executed when the state is entered. There can be more than one of them.

event_action

```
when ( expression ) {
    [statement]...
} state new_state
```

new_state

The name of the new state to enter. This can be the current state.

exit_action

```
exit {
    [statement]...
}
```

`exit` blocks are executed when the state is left. See the options **`-e`** and **`-x`** above 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.

5.5 Statements and Expressions

statement

`{` ***[statement]...*** `}` |
expression `;` |
`if (` ***expression*** `)` ***statement*** |
`else` ***statement*** |
`while (` ***expression*** `)` ***statement*** |
`for (` ***expression*** `;` ***expression*** `;` ***expression*** `)` ***statement*** |
`break;`

As can be seen, most C statements are supported. Strangely, some are missing (but are not hard to add should the need arise).

expression

expression `,` ***expression...*** |
expression binop expression |
expression asgnop expression |
unop expression |
`++` ***expression*** |
`--` ***expression*** |
expression `++` |
expression `--` |
number |
char_const |
string |
name |
name `(` ***expression*** `)` |
expression `[` ***expression*** `]` |
`(` ***expression*** `)`

binop

`-` | `+` | `*` | `/` | `>` | `>=` | `==` | `!=` | `<=` | `<` | `||` | `&&` | `<<` | `>>` | `|` | `^` | `&` | `%` | `?` | `:` | `.` | `->`

These are the usual C binary operators (with the C precendences) with the addition of the “**?**”, “**:**”, “**.**” and “**->**” operators. These can be treated as binary operators because SNL makes no use of the semantics of ternary expressions and structure member access (a side-effect that you may notice is that the state notation compiler will warn that structure tags are unused variables).

asgnop

`=` | `+=` | `-=` | `&=` | `|=` | `/=` | `*=` | `%=` | `<<=` | `>>=` | `^=`

These are the usual C assignment operators.

unop

`+` | `-` | `*` | `&` | `!` | `~`

These are the usual C unary operators.

number
char_const
string
name

The usual C syntax is supported for numbers, character constants, strings and names. Note that, taken together,

expression `,` ***expression...***
name `(` ***expression*** `)`

imply that function calls are permitted (syntactically, the argument list is a comma-separated expression).

5.6 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 their use in escaped C code is subject to special rules.

The term ***variable_name*** refers to any SNL variable that is assigned to a control system variable (or, if it's an array, variables). When using such a variable as a function argument, the function is automatically given access to the details of the underlying control system variable.

Several of these functions are primarily intended to be called only from **`when`** clauses or only from action code. However, unlike in previous versions, it is safe to call any function both in **`when`** clauses and in action code.

`int` function returns should be assumed to be a **`pvStat`** error code unless otherwise spacified.

delay

`int    delay(double` ***delay_in_seconds***`)`

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

The **`-t`** state option ("state_option_stmt" on page 33) controls whether the delay is measured from when the current state was entered from a different state (**`-t`**) or from any state, including itself (**`+t`**, the default)

pvPut

`int    pvPut(`***variable_name***`)`
`int    pvPut(`***variable_name***`,` ***SYNC***`)`
`int    pvPut(`***variable_name***`,` ***ASYNC***`)`

This function puts (or writes) the value of an SNL variable to the underlying control system variable. The function returns the status from the PV layer (*e.g.* **`pvStatOK`** for success).

By default, **`pvPut`** does not wait for the put to be complete; completion must be inferred by other means. The optional **`SYNC`** argument causes it to block on completion with a hard-coded timeout of 10s. The optional **`ASYNC`** argument allows completion to be checked via a subsequent call to **`pvPutComplete`** (typically in a **`when`** clause).

Note that, when using channel access, the **`SYNC`** and **`ASYNC`** arguments result in use of **`ca_put_callback`**; if neither optional argument is specified, **`ca_put`** is called as with previous versions.

pvPutComplete

`int    pvPutComplete(`***variable_name***`)`
`int    pvPutComplete(`***array_name***`)`
`int    pvPutComplete(`***array_name***`, long` ***any***`)`
`int    pvPutComplete(`***array_name***`, long` ***any***`, long *`***pComplete***`)`

This function returns **`TRUE`** if the last put of this control system variable has completed. This call is appropriate only if **`pvPut`**'s optional **`ASYNC`** argument was used.

The first form is appropriate when the SNL variable is a scalar. However, it can also be an array (each of whose elements may be assigned to a different control system variable). In this case, the single argument form returns **`TRUE`** if the last puts of all the elements of the array have completed (the missing arguments are implicitly `0` and `NULL` respectively). If `any` is `TRUE`, then the function returns `TRUE` if any put has completed since the last call. If `pComplete` is non-NULL, it should be a `long` array of the same length as the SNL variable and its elements will be set to `TRUE` if and only if the corresponding put has completed.

pvGet

`int pvGet(`*variable_name*`)`
`int pvGet(`*variable_name*`, `*SYNC*`)`
`int pvGet(`*variable_name*`, `*ASYNC*`)`

This function gets (or reads) the value of an SNL variable from the underlying control system variable. The function returns the status from the PV layer (*e.g.* `pvStatOK` for success). By default, the state set will block until the read operation is complete with a hard-coded timeout of 10s. The asynchronous (`+a`) compile option can be used to prevent this, in which case completion can be checked via a subsequent call to `pvGetComplete` (typically in a `when` clause).

The optional `SYNC` and `ASYNC` arguments override the compile option. `SYNC` blocks and so gives default behavior if `+a` was not specified; `ASYNC` doesn't block and so gives default behavior if `+a` was specified.

pvGetComplete

`int pvGetComplete(`*variable_name*`)`

This function returns `TRUE` if the last get of this control system variable has completed, *i.e.* the value in the variable is current. This call is appropriate only if the asynchronous (`+a`) compile option is specified or `pvGet`'s optional `ASYNC` argument was used.

Unlike `pvPutComplete`, `pvGetComplete` doesn't support arrays.

pvGetQ

`int pvGetQ(`*variable_name*`)`
`int pvGetQ(`*array_name*`)`

This function removes the oldest value from a SNL variable's monitor queue (the variable should have been associated with a queue and an event flag via the `syncQ` statement) and updates the corresponding SNL 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.

If the SNL variable is an array then the behavior is the same regardless of whether the array name or an array element name is specified. This is because a single queue is associated with the entire array.

pvFreeQ

`void pvFreeQ(`*variable_name*`)`

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

As with `pvGetQ`, if the SNL variable is an array then the behavior is the same regardless of whether the array name or an array element name is specified.

pvMonitor

`int pvMonitor(`*variable_name*`)`

This function initiates a monitor on the underlying control system variable.

pvStopMonitor

`int pvStopMonitor(`***variable_name***`)`

This function terminates a monitor on the underlying control system variable.

pvFlush

`void pvFlush()`

This function causes the PV layer to flush its input-output buffer. It just might be needed if performing asynchronous operations *within* an action block (note that the buffer is always flushed on exit from an action block).

pvCount

`int pvCount(`***variable_name***`)`

This function returns the element count associated with the control system variable.

pvStatus

`pvStat pvStatus(`***variable_name***`)`

This function returns the current alarm status for the control system variable (*e.g.* **`pvStatHIHI`**; defined in **`pvAlarm.h`**). The status and severity are only valid after either a **`pvGet`** call has completed or a monitor has been delivered.

pvSeverity

`pvSevr pvSeverity(`***variable_name***`)`

This function returns the current alarm severity (*e.g.* **`pvSevrMAJOR`**). The notes above apply

pvTimeStamp

`TS_STAMP pvTimeStamp(`***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.*

pvAssign

`int pvAssign(`***variable_name, control_system_variable_name***`)`

This function assigns or re-assigned the SNL variable ***variable_name*** to ***control_system_variable_name***. If ***control_system_variable_name*** is an empty string then ***variable_name*** is de-assigned (not associated with any control system variable).

pvAssigned

`int pvAssigned(`***variable_name***`)`

This function returns **`TRUE`** if the SNL variable is currently assigned to a control system variable.

pvConnected

`int pvConnected(`***variable_name***`)`

This function returns **`TRUE`** if the underlying control system variable is currently connected.

pvIndex

`int pvIndex(`***variable_name***`)`

This function returns the index associated with a control system variable. See “User Functions within the State Program” on page 39.

pvChannelCount

`int    pvChannelCount()`

This function returns the total number of control system variables associated with the state program (the term “channel” is a carry-over from the days when the only support message system was channel access).

pvAssignCount

`int    pvAssignCount()`

This function returns the total number of SNL variables in this program that are assigned to underlying control system variables. Note: if all SNL variables are assigned then the following expression is **`TRUE`**:

`pvAssignCount() == pvChannelCount()`

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

pvConnectCount

`int    pvConnectCount()`

This function returns the total number of underlying control system variables that are connected. Note: if all variables 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. It is intended for use within a **`when`** clause.

macValueGet

`char*  macValueGet(char *`***macro_name***`)`

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 **`NULL`**.

5.7 C Compatibility Features

Comments

C-style comments may be placed anywhere in the state program.

Escape to C Code

Because the SNL does not support the full C language, C code may be escaped in the program. The escaped code is not compiled by SNC, but is passed the C 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 pre-processor 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.

User Functions within the State Program

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

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

There is little reason to do this, since a state program can of course be linked against C libraries.

Calling pvGet etc. from C

The built-in SNL functions such as **`pvGet`** cannot be directly used in user-supplied functions. However, most of the built-in functions have a C language equivalent, 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 control system variable name is required, the *index* of that variable must be supplied. This index is obtained via the **pvIndex** function. Furthermore, if the code is 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 40 (this isn't a problem if individual SNL variables are passed as parameters to C code, because the compiler will do the work). 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.

Local variables can be escaped and declared within **`when`** clauses (this will result in a "variable used but not declared" warning from the compiler; ignore it). However, when using the **`+r`** option, the same name cannot be used for SNL and local variables (because the compiler is not clever enough to realize that use of the local variable is intended; see "Variable Modification for Reentrant Option" on page 40). For example:

```
      when ( pvPutComplete( init, TRUE, done ) ) {
%%       long i;
         printf( "init commands not all done:" );
         for ( i = 0; i < N; i++ )
            printf( " %ld", done[i] );
         printf( "\n" );
      } state active
```

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.

Chapter 6: The PV (Process Variable) API

This chapter describes the PV API. It is intended for those who would like to add support for new message systems. It need not be read by those who want to write sequences using message systems that are already supported.

6.1 Introduction

The PV (Process Variable) API was introduced at version 2.0 in order to hide the details of the underlying message system from the sequencer code. Previously, the sequencer code (*i.e.* the modules implementing the sequencer run-time support, not the user-written sequences) called CA routines directly. Now it calls PV routines, which in turn call routines of the underlying message system. This allows new message systems to be supported without changing sequencer code.

6.2 Rationale

Several EPICS tools support both CA and CDEV. They do so in ad hoc ways. For example, **`medm`** uses an **`MEDM_CDEV`** macro and has **`medmCA`** and **`medmCdev`** modules, whereas **`alh`** has an **`alCaCdev`** module that implements the same interface as the **`alCA`** module.

The PV API is an attempt at solving the same problem but in a way that is independent of the tool to which it is being applied. It should be possible to use the PV API (maybe with some backwards-compatible extensions) with **`medm`**, **`alh`** and other CA-based tools. Having done that, supporting another message system at the PV level automatically supports it for all the tools that use the PV API.

Doesn’t this sound rather like the problem that CDEV is solving? In a way, but PV is a pragmatic solution to a specific problem. The PV API is very close in concept to the CA API and is designed to plug in to a CA-based tool with minimal disruption. Why not use the CA API and implement it for other message systems? That could have been done, but would have made the PV API dependent on the EPICS **`db_access.h`** definitions (currently it is dependent only on the EPICS OSI layer).

In any case, a new API was defined and the sequencer code was converted to use it.

6.3 A tour of the API

Overview

The public interface is defined in the file **`pv.h`**, which defines various types such as **`pvStat`**, **`pvSevr`**, **`pvValue`**, **`pvConnFunc`** and **`pvEventFunc`**, then defines abstract **`pvSystem`**, **`pvVariable`** and **`pvCallback`** classes. Finally it defines a C API.

The file **`pv.cc`** implements generic methods (mostly constructors and destructors) and the C API.

Each supported message system **`XXX`** creates a **`pvXxx.h`** file that defines **`xxxSystem`** (extending **`pvSystem`**) and **`xxxVariable`** (extending **`pvVariable`**) classes, and a **`pvXxx.cc`** file that contains the implementations of **`xxxSystem`** and **`xxxVariable`**.

Currently-supported message systems are CA and a Keck-specific one called KTL. The CA layer is very thin (**`pvCa.h`** is 104 lines and **`pvCa.cc`** is 818 lines; both these figures include comments).

The file **`pvNew.cc`** implements a **`newPvSystem`** function that takes a system name argument (*e.g.* “**`ca`**”), calls the appropriate **`xxxSystem`** constructor, and returns it (as a **`pvSystem`** pointer). It would be good to change it to use dynamically-loaded libraries, in which case there would be no direct dependence of the **`pv`** library on any of the **`pvXxx`** libraries (*c.f.* the way CDEV creates **`cdevService`** objects).

Simple C++ PV program (comments and error handling have been removed)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "pv.h"

void event( void *obj, pvType type, int count, pvValue *val,
            void *arg, pvStat stat ) {
    pvVariable *var = ( pvVariable * ) obj;
    printf( "event: %s=%g\n", var->getName(), val->doubleVal[0] );
}

int main( int argc, char *argv[] ) {
    const char *sysNam = ( argc > 1 ) ? argv[1] : "ca";
    const char *varNam = ( argc > 2 ) ? argv[2] : "demo:voltage";

    pvSystem *sys = newPvSystem( sysNam );
    pvVariable *var = sys->newVariable( varNam );

    var->monitorOn( pvTypeDOUBLE, 1, event );
    sys->pend( 10, TRUE );

    delete var;
    delete sys;
    return 0;
}
```

The equivalent program using the C API

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "pv.h"

void event( void *var, pvType type, int count, pvValue *val,
            void *arg, pvStat stat) {
   printf( "event: %s=%g\n", pvVarGetName( var ),
            val->doubleVal[0] );
}

int main( int argc, char *argv[] ) {
   const char *sysNam = ( argc > 1 ) ? argv[1] : "ca";
   const char *varNam = ( argc > 2 ) ? argv[2] : "demo:voltage";
   void *sys;
   void *var;

   pvSysCreate( sysNam, 0, &sys );
   pvVarCreate( sys, varNam, NULL, NULL, 0, &var );

   pvVarMonitorOn( var, pvTypeDOUBLE, 1, event, NULL, NULL );
   pvSysPend( sys, 10, TRUE );

   pvVarDestroy( var );
   pvSysDestroy( sys );
   return 0;
}
```

6.4 The API in More Detail

We will look at the contents of **`pv.h`** (and **`pvAlarm.h`**) in more detail and will specify the constraints that must be met by underlying message systems.

6.4.1 Type definitions

`pv.h` and **`pvAlarm.h`** define various types, described in the following sections.

Status

```
typedef enum {
    pvStatOK           = 0,
    pvStatERROR        = -1,
    pvStatDISCONN      = -2,

    pvStatREAD         = 1,
    pvStatWRITE        = 2,
    .
    pvStatREAD_ACCESS  = 20,
    pvStatWRITE_ACCESS = 21
} pvStat;
```

The negative codes correspond to the few CA status codes that were used in the sequencer. The positive codes correspond to EPICS STAT values.

Severity

```
typedef enum {
```

```
    pvSevrOK      = 0,
    pvSevrERROR   = -1,

    pvSevrNONE    = 0,
    pvSevrMINOR   = 1,
    pvSevrMAJOR   = 2,
    pvSevrINVALID = 3
} pvSevr;
```

These allow easy mapping of EPICS severities.

Data Types

```
typedef enum {
    pvTypeERROR       = -1,
    pvTypeCHAR        = 0,
    pvTypeSHORT       = 1,
    pvTypeLONG        = 2,
    pvTypeFLOAT       = 3,
    pvTypeDOUBLE      = 4,
    pvTypeSTRING      = 5,
    pvTypeTIME_CHAR   = 6,
    pvTypeTIME_SHORT  = 7,
    pvTypeTIME_LONG   = 8,
    pvTypeTIME_FLOAT  = 9,
    pvTypeTIME_DOUBLE = 10,
    pvTypeTIME_STRING = 11
} pvType;

#define PV_SIMPLE(_type) ( (_type) <= pvTypeSTRING )
```

Only the types required by the sequencer are supported, namely simple and “time” types. The “error” type is used to indicate an error in a routine that returns a **`pvType`** as its result.

Data Values

```
typedef char   pvChar;
typedef short  pvShort;
typedef long   pvLong;
typedef float  pvFloat;
typedef double pvDouble;
typedef char   pvString[256]; /* use sizeof( pvString ) */

#define PV_TIME_XXX(_type) \
    typedef struct { \
        pvStat    status; \
        pvSevr    severity; \
        TS_STAMP  stamp; \
        pv##_type value[1]; \
    } pvTime##_type

PV_TIME_XXX( Char   );
PV_TIME_XXX( Short  );
PV_TIME_XXX( Long   );
PV_TIME_XXX( Float  );
PV_TIME_XXX( Double );
PV_TIME_XXX( String );

typedef union {
    pvChar       charVal[1];
```

```
    pvShort      shortVal[1];
    pvLong       longVal[1];
    pvFloat      floatVal[1];
    pvDouble     doubleVal[1];
    pvString     stringVal[1];
    pvTimeChar   timeCharVal;
    pvTimeShort  timeShortVal;
    pvTimeLong   timeLongVal;
    pvTimeFloat  timeFloatVal;
    pvTimeDouble timeDoubleVal;
    pvTimeString timeStringVal;
} pvValue;

#define PV_VALPTR(_type,_value) \
    ( ( PV_SIMPLE(_type) ? \
        ( void * ) ( _value ) : \
        ( void * ) ( &_value->timeCharVal.value ) ) )
```

`pvValue` is equivalent to **`db_access_val`** and, like it, is not self-describing (remember, the idea is that the PV layer is a drop-in replacement for CA).

Obviously, the introduction of **`pvValue`** means that values must be converted between it and the message system's internal value representation. This is a performance hit but one that was deemed worthwhile given that there is currently no appropriate "neutral" (message system independent) value representation. Once the replacement for GDD is available, it will maybe be used in preference to **`pvValue`**.

Callbacks

```
typedef void (*pvConnFunc)( void *var, int connected );

typedef void (*pvEventFunc)( void *var, pvType type, int count,
                             pvValue *value, void *arg, pvStat status );
```

In both cases, the **`var`** argument is a pointer to the **`pvVariable`** that caused the event. It is passed as a **`void*`** so that the same function signature can be used for both C and C++. In C, it would be passed to one of the **`pvVarXxx`** routines; in C++ it would be cast to a **`pvVariable*`**.

`pvConnFunc` is used to notify the application that a control system variable has connected or disconnected

- **`connected`** is 0 for disconnect and 1 for connect

`pvEventFunc` is used to notify an application that a get or put has completed, or that a monitor has been delivered

- **`type`**, **`count`** and **`arg`** come from the request
- **`value`** is of type **`type`** and contains **`count`** elements
    - it may be NULL on put completion (the application should check)
    - it might also be NULL if **`status`** indicates failure (the application should check)
    - it is filled with zeroes if the control system variable has fewer than **`count`** elements
- **`status`** comes from the underlying message system
    - it is converted to a **`pvStat`**

6.4.2 pvSystem Class

`pvSystem` is an abstract class that must be extended by specific message systems. An application typically contains a single instance, created by **`newPvSystem`** as described in “Overview” on page 42. There’s nothing to stop an application having several instances, each corresponding to a different message system, but the sequencer doesn’t do this. Also, there is no way to pend on events from a set of **`pvSystem`**s.

Refer to **`pv.h`** for explicit detail. The following sections describe various important aspects of the class.

Variable Creation

The **`newVariable`** method creates a new **`pvVariable`** corresponding to the same message system as the calling **`pvSystem`**. It should be used in preference to the concrete **`xxxVariable`** constructors since it doesn’t require knowledge of **`xxx`**!

Event Handling

The **`flush`** and **`pend`** methods correspond to **`ca_flush`**, **`ca_pend_io`** and **`ca_pend_event`** (the latter two are combined into a single **`pend`** method with an optional **`wait`** argument; **`wait=FALSE`** gives **`ca_pend_io`** behavior, *i.e.* exit when pending activity is complete, and **`wait=TRUE`** gives **`ca_pend_event`** behavior, *i.e.* wait until timer expires).

Locking

The **`lock`** and **`unlock`** methods take and give a (recursive) mutex that can be used to prevent more than one thread at a time from being within message system code. This is not necessary for thread-safe message systems such as CA.

Debugging

A **`debug`** flag is supported (it’s an optional argument to the constructor and to the **`newVariable`** method) and is used to report method entry, arguments and other information. Debug flags are used consistently throughout the entire PV layer.

Error Reporting

A message system-specific status, a severity (**`pvSevr`**), a status (**`pvStat`**), and an error message, are maintained in member variables. The concrete implementations should use the provided accessor functions to maintain up-to-date values for them. The **`pvVariable`** class supports the same interface.

6.4.3 pvVariable Class

`pvVariable` is an abstract class that must be extended by specific message systems. It corresponds to a control system variable accessed via its message system. Each **`pvVariable`** object is associated with a **`pvSystem`** object that manages system-wide issues like locking and event handling.

Refer to **`pv.h`** for explicit detail. The following sections describe various important aspects of the class.

Creation

The constructor specifies the corresponding **`pvSystem`**, the variable name (which is copied), an optional connection function, an optional private pointer, and an optional debug flag (0 means to inherit it from the **`pvSystem`**).

The constructor should initiate connection to the underlying control system variable and should arrange to call the connection function (if supplied) on each connect or disconnect.

Reading

Like CDEV, the PV API supports the following **`get`** methods:

```
pvStat get( pvType type, int count, pvValue *value );
pvStat getNoBlock( pvType type, int count, pvValue *value );
pvStat getCallback( pvType type, int count, pvEventFunc func,
                    void *arg = NULL );
```

- **`get`** blocks on completion for a message system specific timeout (currently 5s for CA)
- **`getNoBlock`** doesn't block: the value can be assumed to be valid only if a subsequent **`pend`** (with **`wait=FALSE`**) returns without error (currently, the CA implementation of **`getNoBlock`** does in fact block; it should really use **`ca_get_callback`**; note, however, that this is not an issue for the sequencer because it is not used).
- **`getCallback`** calls the user-specified function on completion; there is no timeout

Writing

Like CDEV, the PV API supports the following put methods:

```
pvStat put( pvType type, int count, pvValue *value );
pvStat putNoBlock( pvType type, int count, pvValue *value );
pvStat putCallback( pvType type, int count, pvValue *value,
                    pvEventFunc func, void *arg = NULL );
```

- **`put`** blocks on completion for a message system specific timeout (currently 5s for CA; note that CA does not call **`ca_put_callback`** for a blocking put)
- **`putNoBlock`** doesn't block: successful completion can be inferred only if a subsequent **`pend`** (with **`wait=FALSE`**) returns without error (note that CA does not call **`ca_put_callback`** for a non-blocking put)
- **`putCallback`** calls the user-specified function on completion; there is no timeout (note that CA calls **`ca_put_callback`** for a put with callback)

Monitoring

The PV API supports the following monitor methods:

```
pvStat monitorOn( pvType type, int count, pvEventFunc func,
                  void *arg = NULL, pvCallback **pCallback = NULL );
pvStat monitorOff( pvCallback *callback = NULL );
```

- **`monitorOn`** enables monitors; when the underlying message system posts a monitor, the user-supplied function will be called (CA enables **`value`** and **`alarm`** monitors)
- **`monitorOff`** disables monitors; it should be supplied with the callback value that was optionally returned by **`monitorOn`**
- some message systems will permit several **`monitorOn`** calls for a single variable (CA does); this is optional (the sequencer only ever calls it once per variable)
- all message systems must permit several **`pvVariable`**s to be associated with the same underlying control system variable and, when a monitor is posted, must guarantee to propagate it to all the associated **`pvVariable`**s

Miscellaneous

`pvVariable` supports the same debugging and error reporting interfaces as **`pvSystem`**.

6.5 Supporting a New Message System

CDEV is an obvious message system to support. This section should provide the necessary information to support it or another message system. It includes an example of a partly functional **`file`** message system.

Note that file names in this section are assumed to be relative to the top of the sequencer source tree.

6.5.1 Check-list

This section gives a check-list. See Section 6.5.2 on page 48 for an example of each stage.

Create New Files

For message system XXX, the following files should be created:

- **`src/pv/pvXxx.h`**, definitions
- **`src/pv/pvXxx.cc`**, implementation

Edit src/pv/pvNew.cc

Edit **`src/pv/pvNew.cc`** according to existing conventions. Assume that the **`PVXXX`** pre-processor macro is defined if and only if support for XXX is to be compiled in. See “src/pv/pvNew.cc” on page 50 for an example.

Edit configure/ RELEASE

By convention, the **`configure/RELEASE`** file defines the various **`PVXXX`** make macros. See “configure/RELEASE” on page 51 for an example.

Edit src/pv/Makefile

By convention, XXX support should be compiled only if the **`PVXXX`** make macro is defined and set to **`TRUE`**. See “pv/src/Makefile” on page 51 for an example.

Edit application Makefiles

Edit application **`Makefile`**s to search the **`pvXxx`** library and any other libraries that it references. It is, unfortunately, necessary, to link applications against all message systems. This is because **`src/pv/pvNew.cc`** references them all. This problem will disappear if and when **`pvNew`** is changed to load **`pvXxx`** libraries dynamically by name. See “test/pv/ Makefile” on page 51 for an example.

6.5.2 Example

As an example, we consider a notional **`file`** message system with the following attributes:

- Commands are read from file **`fileI`**; they are of the form “**`keyword value`**”, *e.g.* “**`fred 2`**” sets variable **`fred`** to **`2`** (
- Results are written to file **`fileO`**; they are of the same form as the commands
- Everything is a string

The files **`pvFile.h`** and **`pvFile.cc`** can be found in the **`src/pv`** directory. They compile and run but do not implement full functionality (left as an exercise for the reader!).

src/pv/pvFile.h

Only some sections of the file are shown.

```
class fileSystem : public pvSystem {
```

```
public:
   fileSystem( int debug = 0 );
   ~fileSystem();

   virtual pvStat pend( double seconds = 0.0, int wait = FALSE );

   virtual pvVariable *newVariable( const char *name,
      pvConnFunc func = NULL, void *priv = NULL, int debug = 0 );

private:
   FILE *ifd_;
   FILE *ofd_;
   fd_set readfds_;
};

class fileVariable : public pvVariable {

public:
    fileVariable( fileSystem *system, const char *name, pvConnFunc
                func = NULL, void *priv = NULL, int debug = 0 );
    ~fileVariable();

    virtual pvStat get( pvType type, int count, pvValue *value );
    virtual pvStat getNoBlock( pvType type, int count,
                            pvValue *value );
    virtual pvStat getCallback( pvType type, int count, pvEventFunc
                             func, void *arg = NULL );
    virtual pvStat put( pvType type, int count, pvValue *value );
    virtual pvStat putNoBlock( pvType type, int count, pvValue
                            *value );
    virtual pvStat putCallback( pvType type, int count, pvValue
          *value, pvEventFunc func, void *arg = NULL );
    virtual pvStat monitorOn( pvType type, int count, pvEventFunc
          func, void *arg = NULL, pvCallback **pCallback = NULL );
    virtual pvStat monitorOff( pvCallback *callback = NULL );

    virtual int getConnected() const { return TRUE; }
    virtual pvType getType() const { return pvTypeSTRING; }
    virtual int getCount() const { return 1; }

private:
    char *value_;       /* current value */
};
```

src/pv/pvFile.cc

Most of the file is omitted.

```
fileSystem::fileSystem( int debug ) :
   pvSystem( debug ),
   ifd_( fopen( "iFile", "r" ) ),
   ofd_( fopen( "oFile", "a" ) )
{
    if ( getDebug() > 0 )
       printf( "%8p: fileSystem::fileSystem( %d )\n", this, debug);

   if ( ifd_ == NULL || ofd_ == NULL ) {
      setError( -1, pvSevrERROR, pvStatERROR, “failed to open ”
               “iFile or oFile” );
```

```
       return;
    }

    // initialize fd_set for select()
   FD_ZERO( &readfds_ );
   FD_SET( fileno( ifd_ ), &readfds_ );
}

pvStat fileVariable::get( pvType type, int count, pvValue *value )
{
   if ( getDebug() > 0 )
      printf( "%8p: fileVariable::get( %d, %d )\n", this, type,
              count );

   printf( "would read %s\n", getName() );
   strcpy( value->stringVal[0], "string" );
   return pvStatOK;
}

pvStat fileVariable::put( pvType type, int count, pvValue *value )
{
   if ( getDebug() > 0 )
      printf( "%8p: fileVariable::put( %d, %d )\n", this, type,
              count );

   printf( "would write %s\n", getName() );
   return pvStatOK;
}
```

src/pv/pvNew.cc

Edit this to support the **`file`** message system. Some parts of the file are omitted.

```
#include "pv.h"

#if defined( PVCA )
#include "pvCa.h"
#endif

#if defined( PVFILE )
#include "pvFile.h"
#endif

pvSystem *newPvSystem( const char *name, int debug ) {

#if defined( PVCA )
   if ( strcmp( name, "ca" ) == 0 )
      return new caSystem( debug );
#endif

#if defined( PVFILE )
   if ( strcmp( name, "file" ) == 0 )
      return new fileSystem( debug );
#endif

return NULL;
}
```

configure/RELEASE

Edit this to support the **`file`** message system. Comment out these lines to disable use of message systems. Some parts of the file are omitted.

```
PVCA = TRUE
PVFILE = TRUE
```

pv/src/Makefile

Edit this to support the **`file`** message system. Some parts of the file are omitted.

```
LIBRARY += pv
pv_SRCS += pvNew.cc pv.cc

ifeq "$(PVCA)" "TRUE"
USR_CPPFLAGS      += -DPVCA
INC               += pvCa.h
LIBRARY           += pvCa
pv_SRCS_vxWorks   += pvCa.cc
pvCa_SRCS_DEFAULT += pvCa.cc
endif

ifeq "$(PVFILE)" "TRUE"
USR_CPPFLAGS += -DPVFILE
INC          += pvFile.h
LIBRARY      += pvFile
pvFile_SRCS  += pvFile.cc
endif
```

test/pv/Makefile

This includes rules for building the test programs of Section 6.3 on page 42. Only those rules are shown.

```
TOP = ../..
include $(TOP)/configure/CONFIG

PROD = pvsimpleCC pvsimpleC

PROD_LIBS += seq pv
seq_DIR    = $(SUPPORT_LIB)

ifeq "$(PVFILE)" "TRUE"
PROD_LIBS += pvFile
endif

ifeq "$(PVCA)" "TRUE"
PROD_LIBS += pvCa ca
endif

PROD_LIBS += Com

include $(TOP)/configure/RULES
```

Chapter 7: Examples of State Programs

7.1 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
    }
}
```

7.2 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 {
      ...
   }
}
```

7.3 Complex example

This example needs updating.

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 2.3 on page 8. 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;

int       fd;       /* file descriptor for logging */
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");
            fd = 1;
         }
         else
         {
            fd = open(pmac, (O_CREAT | O_WRONLY), 0664);
            if (fd == ERROR)
            {
               printf(“Can’t open %s\n”, pmac);
               exit (-1);
            }
         }
         fdprintf(fd, “Starting 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);
}
}%
```

Chapter 8: Installation

The sequencer is distributed as an EPICS R3.14 **`makeBaseApp`** application. This chapter describes how to obtain, unpack, build, install, verify and use the distribution.

8.1 Prerequisites

EPICS R3.14 (any version) or later must be installed on your system.

8.2 Obtaining the distribution

The distribution should be obtained via the sequencer home page which is, at the time of writing, at URL **`http://www2.keck.hawaii.edu:3636/realpublic/epics/seq`** (what a mouthful). This describes the available versions and will point you to a gzipped tar file with a name of the form **`seq-n.m.p.tar.gz`** (**`n.m.p`** is the version number, *e.g.* **`2.0.0`**).

Select and download the appropriate version. In what follows, we will assume that you downloaded v2.0.0. However, the instructions will apply to this or any later version.

Note that, from v2.0.0, the third digit is the patch level and will be incremented each time a new version is released, no matter how minor the changes. The second digit is the minor version number and will be incremented each time functional changes are made. The first digit is the major version number and will be incremented only when major changes are made.

8.3 Unpacking the distribution

`cd` to the directory that you wish to be the parent of the sequencer tree. Then unpack and untar the file. For example (these steps can be combined by clever use of pipes using syntax that I can never remember, or else if you have GNU tar, its **`-z`** option will decompress on the fly).

```
% gunzip seq-2.0.0.tar.gz
% tar xvf seq-2.0.0.tar
```

This creates a directory tree with the following general structure (this is part of the file in the top-level directory).

```
/sequencer

   README           This file (general notes at the top, followed
                    by release notes, most recent first)

(etc... update when README has been updated!)
```

8.4 Preparing to build

It will be necessary to edit the files **`configure/RELEASE`** and **`configure/CONFIG`** before building. Here are copies of these files, with the lines that you are likely to have change highlit:

```
#RELEASE Location of external products
EPICS_BASE=/home/wlupton/epics/anl/base
TEMPLATE_TOP=$(EPICS_BASE)/templates/makeBaseApp/top
SEQ=/home/wlupton/epics/seq

#CONFIG
include $(TOP)/configure/CONFIG_APP
# Add any changes to make rules here

#CROSS_COMPILER_TARGET_ARCHS = vxWorks-68040
CROSS_COMPILER_TARGET_ARCHS =

# shareable library version (from CONFIG_BASE)
SHRLIB_VERSION = $(EPICS_VERSION).$(EPICS_REVISION)

# sequencer version number (replaces old Version file)
SEQ_VERSION = 2.0.0

# override to use snc from SEQ
SNC = $(SEQ)/bin/$(EPICS_HOST_ARCH)/snc

# which message systems to support (comment to disable)
PVCA = TRUE
#PVFILE = TRUE
#PVKTL = TRUE
```

In **`RELEASE`**, you should select EPICS base via the **`EPICS_BASE`** macro and the top of the sequencer tree via the **`SEQ`** macro.

In **`CONFIG`**, you should select the target architectures for which to build via the **`CROSS_COMPILER_TARGET_ARCHS`** macro (a subset of those for which EPICS has been built), and the message systems to support via the **`PVXXX`** macros.

8.5 Building and installing

Ensure that your environment is configured for building EPICS applications. The only EPICS requirement is that the **`EPICS_HOST_ARCH`** environment variable be set correctly (you can use the **`$EPICS_BASE/startup/EpicsHostArch`** script to set it). However, if you built EPICS with shareable library support, your **`LD_LIBRARY_PATH`** environment variable will have to include **`$EPICS_BASE/lib/$EPICS_HOST_ARCH`**, and if you are using gcc with shareable library support, it will have to include the directory that contains **`libstdc++.so`**. These notes are written from a Solaris standpoint; details may vary slightly under other architectures.

`cd` to the top of the sequencer tree and run GNU make. The tree should build without incident. Please feed back any build problems (and their resolutions!) to me. My e-mail address is on the front cover of this manual.

Note that make builds in the **`configure`** directory, then the **`src`** tree, and finally the **`test`** tree. A failure in the **`test`** tree will not impact your ability to write sequences.

8.6 Verifying the installation

Under Solaris, the **`-R`** loader option will have been used to link executables, so **`LD_LIBRARY_PATH`** should need no further additions. Under other operating systems, it may be necessary to append **`$SUPPORT/lib/$EPICS_HOST_ARCH`**, where **`SUPPORT`** has the value that you gave it in **`configure/RELEASE`**.

`cd` to **`$SUPPORT/bin/$EPICS_HOST_ARCH`**. It should look like this:

```
demo          sncDelay      sncEntryVar   sncExample    sncOpttVar
demo.vws      sncEntry      sncExEntry    sncExitOptx
snc           sncEntryOpte  sncExOpt      sncOptt
```

Try running demo. This includes its own CA server, so no IOC or portable CA server is needed. It should look something like this:

```
% ./demo
Starting iocInit
############################################################################
###  @(#)EPICS IOC CORE built on Mar 22 2000
###  @(#)Version R3.14.0.alpha $$Date: 2002/05/15 23:59:31 $$
###  @(#)Built date Mar 22 2000
############################################################################
db_attach_pvAdapter I dont know what to call
iocInit: All initialization complete
@(#)SEQ Version 2.0.0: Fri Mar 31 16:50:09 HST 2000
osiSockDiscoverInterfaces(): ignoring loopback interface: lo0
osiSockDiscoverInterfaces(): net intf hme0 found
osiSockDiscoverInterfaces(): ignoring loopback interface: lo0
osiSockDiscoverInterfaces(): net intf hme0 found
Spawning thread 0xa54c0: "demo_1"
```

```
Spawning thread 0xa5580: "demo_2"
Spawning state program "demo", thread 0x98240: "demo"
demo_1 2000/03/31 17:14:24: start -> ramp_up
demo 2000/03/31 17:14:31: light_off -> light_on
demo_1 2000/03/31 17:14:35: ramp_up -> ramp_down
```

If you see the "**`start -> ramp_up`**" *etc.* messages, things are good. If not, some channels haven't connected (use the "**`–`**" to command to find out which).

Issue the "**`i`**" command. You should something like this:

```
i
            NAME        ID       PRI    STATE      WAIT
          _main_    2adf8         0       OK
          errlog    323a0        10       OK
          taskwd    32508        10       OK
           cbLow    3cf48        59       OK
        cbMedium    30e10        64       OK
          cbHigh    30ed8        71       OK
        dbCaLink    313f0        50       OK
     CAC process    31570        50       OK
        scanOnce    4c5f0        70       OK
          scan10    4c878        60       OK
           scan5    4c908        61       OK
           scan2    4c9b0        62       OK
           scan1    4ca40        63       OK
         scan0.5    4cad0        64       OK
         scan0.2    4cb60        65       OK
         scan0.1    4cbf0        66       OK
           CAtcp    4cda0        20       OK
           CAudp    4ce30        19       OK
          seqAux    4def0        51       OK
        CAonline    4df80         7       OK
     CAC process    8ee70        50       OK
            demo    9b310        50       OK
    CAC UDP Recv    9c588        10       OK
    CAC UDP Send    9c618        10       OK
  osiTimerQueue     9c6e0         0       OK
          demo_1    9cac0        50       OK
          demo_2    9cb50        50       OK
    CAC TCP Recv    9d070        10       OK
    CAC TCP Send    9d100        10       OK
        CA event    b2cd8        19       OK
        CAclient    b2d68        10       OK
```

Finally, go to an xterm and do the following:

```
% caget ss0
ss0                            light
% caget ss1
ss1                            ramp
```

This illustrates the very basic "sequencer device support" in this release. These records are returning the names of the first two state-sets of the above sequence.

Most (maybe all) of the other test programs do not connect to control system variables and can be run without an IOC. For example:

```
% sncExitOptx
@(#)SEQ Version 2.0.0: Fri Mar 31 16:50:09 HST 2000
Spawning state program "sncexitoptx", thread 0x30868: "sncexitoptx"
```

```
low, delay timeout, incr v and now reenter low
v = 1
Pause on each exit of low, including 'iterations'
low, delay timeout, incr v and now reenter low
v = 2
Pause on each exit of low, including 'iterations'
low, delay timeout, incr v and now reenter low
v = 3
Pause on each exit of low, including 'iterations'
^D
```

8.7 Using the installation

This section assumes that you are working in a **`makeBaseApp`** environment. The more general information in Section 3.8 on page 22 should help if this is not the case.

You need to edit your own **configure/RELEASE** and **configure/CONFIG** files to reference the correct sequencer version. The necessary changes are similar to those that were made to build the sequencer (Section 8.4 on page 60). Here are versions of the files with the lines that must be added for the sequencer highlit:

```
#RELEASE Location of external products
EPICS_BASE=_EPICS_BASE_
TEMPLATE_TOP=_TEMPLATE_TOP_
SEQ=_SEQ_TOP_

#CONFIG
include $(TOP)/configure/CONFIG_APP
# Add any changes to make rules here

#CROSS_COMPILER_TARGET_ARCHS = vxWorks-68040

SNC = $(SEQ)/bin/$(EPICS_HOST_ARCH)/snc
```

You can refer to the various **`Makefile`**s in the test tree to see how to write your own sequences. For example, here is the **`Makefile`** for the above **`sncExitOptx`** program (**`test/validate/Makefile`**):

```
TOP = ../..
include $(TOP)/configure/CONFIG

SNCFLAGS = +m

SEQS = sncDelay sncEntry sncEntryOpte sncEntryVar sncExitOptx \
       sncOptt sncOpttVar
PROD = $(SEQS)
OBJS_vxWorks = $(SEQS)

PROD_LIBS += seq
seq_DIR    = $(SEQ_LIB)

include $(TOP)/test/Makefile.pv

PROD_LIBS += Com

include $(TOP)/configure/RULES
```

This **`Makefile`** includes **`test/Makefile.pv`** but you will probably want to look at that and bring what you need inline (it handles all the possible message systems and you will likely be using only a single message system). For example, to use CA, the following would be fine.

```
TOP = ../..
include $(TOP)/configure/CONFIG

SNCFLAGS = +m

SEQS = sncDelay sncEntry sncEntryOpte sncEntryVar sncExitOptx \
       sncOptt sncOpttVar
PROD = $(SEQS)
OBJS_vxWorks = $(SEQS)

PROD_LIBS += seq pv pvCa ca Com
seq_DIR    = $(SEQ_LIB)

include $(TOP)/configure/RULES
```

The only real requirements here are that **`SEQ`** is defined to point to the head of the tree in which the sequencer has been installed (see **`configure/RELEASE`**; **`SEQ_LIB`** is automatically defined by a make rule in **`configure`**), and that **`SNC`** is defined like this (see **`configure/CONFIG`**):

```
SNC = $(SUPPORT)/bin/$(EPICS_HOST_ARCH)/snc
```

Chapter 9: Acronyms/Glossary

This is not yet a terribly useful section.

9.1 Acronym List

API	Application Programming Interface
CA	Channel Access
CDEV	Control DEVice
EPICS	Experimental Physics and Industrial Control System
GDD	Generalized Device Descriptor
IOC	Input/Output Controller
KTL	Keck Task Library
OSI	Operating System Independent
PV	Process Variable
SNC	State Notation Compiler
SNL	State Notation Language
STD	State Transition Diagram
UML	Unified Modeling Language

9.2 Glossary

Channel Access	EPICS software that supports network independent access to IOC databases.
Control DEVice	API (originating at Jefferson Lab) that provide message system independent means of interacting with an underlying control system.
Input/Output Controller	The VME/VXI based chassis containing a real-time processor, various I/O modules, and VME modules that provide assess to other I/O buses such as GPIB.

Index

A

arrays . . . 11
assign . . . 4, 8, 10, 12

C

C
  blocks of C code . . . 39
  calling pvGet etc. from C . . . 39
  comments . . . 38
  escape to C . . . 39
  inline C code . . . 39
  local variables . . . 39
  variable extent . . . 39
C pre-processor . . . 20
CA . . . 1–3, 19, 22, 35, 38, 41
  ca_put . . . 35
  ca_put_callback . . . 35
channel access, see CA

D

debugging
  seqChanShow . . . 3, 6, 26
  seqQueueShow . . . 3, 26
  seqShow . . . 3, 6, 26
  under Unix . . . 26
delay . . . 10, 33, 35
device support . . . 3

E

efClear . . . 38
efSet . . . 38
efTest . . . 38
efTestAndClear . . . 4, 38
entry handler . . . 30
  one-off . . . 3, 28, 30
  per-state . . . 3, 33
errlogPrintf . . . 5, 28
event flags . . . 13
  synchronizing with variables . . . 32
exit handler . . . 30
  one-off . . . 3, 30
  per-state . . . 3, 33

F

files
  epicsTypes.h . . . 19
  errlog.h . . . 19
  iocCoreLibrary . . . 19
  libca . . . 19
  libCom . . . 19
  libpv . . . 19
  libpvCa . . . 19
  libseq . . . 19
  osiSem.h . . . 19
  osiThread.h . . . 19
  pv.cc . . . 42
  pv.h . . . 42–43, 46
  pvAlarm.h . . . 19, 37, 43
  pvCa.cc . . . 42
  pvCa.h . . . 42
  pvLibrary . . . 19
  pvNew.cc . . . 42
  seqCom.h . . . 19
  seqLibrary . . . 19
  shareLib.h . . . 19
  snc . . . 19
  taskwd.h . . . 19
  tsStamp.h . . . 19
functions
  delay . . . 10, 33, 35
  efClear . . . 38
  efSet . . . 38
  efTest . . . 38
  efTestAndClear . . . 4, 38
  errlogPrintf . . . 5, 28
  iocLogInit . . . 28
  macValueGet . . . 27, 38
  newPvSystem . . . 46
  pvAssign . . . 37
  pvAssignCount . . . 12, 38
  pvAssigned . . . 12, 37
  pvChannelCount . . . 12, 38
  pvConnectCount . . . 12, 38
  pvConnected . . . 16, 37
  pvCount . . . 16, 37
  pvFlush . . . 37
  pvFreeQ . . . 3, 14, 36
  pvGet . . . 4, 14–15, 28, 36
  pvGetComplete . . . 14, 35–36
  pvGetQ . . . 3, 14, 32, 36
  pvIndex . . . 37
  pvMonitor . . . 36
  pvPut . . . 4, 8, 10–12, 28, 35
  pvPutComplete . . . 4, 15, 35
  pvSeverity . . . 12, 37
  pvStatus . . . 12, 37
  pvStopMonitor . . . 37

- pvTimeStamp 12, 37
- seq. 30
- seqLog 28
- seqStop 30

I
iocLogInit 28

K
Kozubal, Andy 1
KTL 42

L
Lupton, William 1

M
macValueGet. 27, 38
monitor 8
monitors
- de-queueing 36
- queuing 14, 32

N
name 30
newPvSystem 46

O
Operating System Independent, see OSI
OSI 1, 3

P
parameters
- debug 27
- logfile 27–28
- name 27, 30
- priority 28
- stack 28

process variable, see PV API
program. 9, 27
PV API . 2–3, 16, 27, 35–37, 41, 45–47
pv.cc 42
pv.h 42–43, 46
pvAlarm.h. 37, 43
pvAssign. 37
pvAssignCount 12, 38
pvAssigned 12, 37
pvCa.cc 42
pvCa.h. 42
pvChannelCount 12, 38
pvConnectCount 12, 38
pvConnected 16, 37
pvCount 16, 37
pvFlush. 37
pvFreeQ 3, 14, 36
pvGet 4, 14–15, 28, 36
pvGetComplete 14, 35–36
pvGetQ. 3, 14, 32, 36
pvIndex 37
pvMonitor 36
pvNew.cc 42
pvPut 4, 8, 10–12, 28, 35
pvPutComplete. 4, 15, 35
pvSeverity 12, 37
pvStatus 12, 37
pvStopMonitor. 37
pvTimeStamp. 12, 37

R
running away to the wide blue yonder39

S
seq 30
seqChanShow. 3, 6, 26
seqLog 28
seqQueueShow. 3, 26
seqShow 3, 6, 26
seqStop. 4, 26–27, 30
sequencer
- device support 3

sequences
- creation. 25
- deletion. 26
- event flags 13

SNC 19
SNL 19
- compiler options
  - a 4, 14, 20, 36
  - c 16, 20
  - d 20
  - e 21, 28
  - l 21
  - m 3, 19, 21
  - r 16, 21, 32, 39–40
  - w. 21
- option placement 21
- options in source files. 32
- state options
  - e 3, 33
  - t 3, 33, 35
  - x 3, 33
- structure 8
- syntax 8

State Notation Compiler, see SNC
State Notation Language, see SNL
state sets 9
state transition diagram, see STD
statements
- assign 4, 8, 10
- monitor. 8
- program 9, 27
- sync 4, 13
- syncQ. 3–4, 36
- when 4, 8, 10, 13–14, 16, 28, 33, 35–36, 38–39

STD 7
sync 4, 13
synchronization 13
syncQ. 3–4, 36

T
types. 11

V
variables
- arrays 11
- assigning 31
- assignment to. 12
- asynchronous get. 36
- asynchronous put. 35
- de-assignment from 12
- declaration of. 30
- de-queueing monitors 36
- emptying queues 36
- extent 39
- local 39
- monitoring 31
- queuing monitors. 14, 32
- status of 12
- synchronizing with 13
- synchronizing with event flags 13, 32
- synchronous get. 36
- synchronous put. 35
- types. 11
- undeclared warnings 39

VxWorks 3

W
when4, 8, 10, 13–14, 16, 28, 33, 35–36, 38–39
White, Greg 1