State Notation Language and Sequencer Users' Guide

Release 2.0.99

William Lupton (wlupton@keck.hawaii.edu)
Benjamin Franksen (benjamin.franksen@bessy.de)

June 16, 2010

CONTENTS

Warning: The documentation has not yet been completely updated to cover the latest release. Also, the conversion to reStructuredText is not yet complete in all places, some formatting may be odd, etc. I am working on it. If in doubt, check the release notes. If still in doubt, send me an email and I will try to answer your questions.

CHAPTER

ONE

INTRODUCTION

1.1 About

This project is a component of the Experimental Physics and Industrial Controls System, short EPICS. EPICS is a system of interactive applications, development tools, and a common run-time environment that allows users to build and execute real-time control and data acquisition systems for experimental facilities, such as particle accelerators and telescopes.

The state notation language *SNL* allows programming sequential operations that interact with EPICS process variables without the usual complexity involved with task scheduling, semaphores, event handling, and I/O programming. The language is a restricted subset if C, enhanced with extra syntax for descibing state sets, states, state transitions, and the binding of program variables to named process variables. See the *Tutorial* for a gentle introduction.

From the SNL source code, the SNL compiler *snc* generates a number of C code procedures that are called by the run-time sequencer, a C library contained in this project. This library handles all low-level management of connecting, monitoring, and changing process variables, as well as task (thread) creation and communication. 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.

1.2 Acknowledgements

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

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

1.3 Copyright

1.4 Note on Versions

This text is supposed to describe version 2.1. Version 2.1 adds support for local definitions (including variable declarations) at all levels, a significantly improved compiler that employs new lexer and parser generators, and lots of bugfixes. See Notes on Release 2.1 for details.

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

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

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

1.4.1 Versioning Policy

Starting with 2.0.0, the third digit is the patch level and will be incremented each time a new version is released, no matter how minor the changes. 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.

1.5 Notes on Release 2.1

Most of the changes relative to 2.0.12 are to the SNL compiler, but a small number of changes have been made to the runtime library as well.

The extensions are mostly conservative: existing SNL programs should compile and work just as with 2.0.12 (with one exception, see next paragraph). This is not easy to guarantee, however, as there are many corner cases where the manual is imprecise and the code was convoluted (and possibly erroneous), especially with regard to the `syncQ` feature.

There is one (mis-)feature I have removed: 2.0.x allowed more than one entry or exit block inside the same state. This has no semantic value, the action statements are simply concatenated as if they had been written in one block.

So if you (for whatever reason) relied on this, then for each state you'll have to merge all its entry blocks into one (and similar for its exit blocks).

For 2.1, the documentation has been converted to reStructuredText. We use Sphinx (http://sphinx.pocoo.org/) to generate web pages from the rst source files. What Sphinx does to a hand full of drab (almost) plain text files is simply phantastic. Thanks and Kudos to the creators of Sphinx!

1.5.1 New Features

The most important extensions are local definitions and the new state change command. Suggestions, criticism, or encouragements regarding these extensions are welcome, of course. (Send them to tech-talk@aps.anl.gov, core-talk@aps.anl.gov, or benjamin.franksen@bessy.de).

Local Definitions

Here, "definitions" is to be understood as in the *SNL reference*, i.e. `option``s`, `variable declarations`, ```assign`, `monitor`, `sync`, and `syncQ` constructs. These definitions have to appear (in any order) right after the opening brace and before any other content (code, states,transitions), similar as in C. However, not every definition is allowed everywhere:

- `option` definitions are restricted just as before, i.e. at the top level (for program options) and inside a state (for state options)
- `assign`, `monitor`, `sync`, and `syncQ` can appear inside a state set (`ss <state_set_name> {...}`) and inside a state (`state <state_name> {...}`), in addition to the top level
- foreign declarations (see below) and event flag declarations are restricted to the top-level
- variable declarations can appear at the start of any block (state set, state, transition, entry, exit, and compound statement blocks); their scope is always limited (statically) to the smallest enclosing block

Local variable declarations come in two flavours, depending on where they appear:

1. Variables of *unlimited life time* are global variables and those which are local to a state set or a state clause. Only variables of this sort can be assigned to a process variable, monitored, synced etc.
2. Variables declared in any other block have lifetime *limited to the enclosing block*, they disappear when the block exits, just as block local variables in C.

Variable declarations are restricted to the small set of types offered by SNL just as before. Scalar variable declarations may be initialized with an arbitrary expression (for top-level variables the C compiler will only allow constant expressions, but this isn't checked by the SNL compiler).

State Change Command

This is an experimental feature. It adds a new primitive action statement

```
state <state-name>;
```

Its operational meaning is to immediately return from the enclosing transition action block and to enter the named state, instead of the default state that is given after the block as before. Entry and exit blocks are respected exactly as with all other state changes.

I have termed this an experimental feature because I am not sure it is good to offer something like that. It is certainly similar to a "goto", in that it enables unstructured control flow. I am interested in your opinion!

Minor Extensions/Improvements

- You can avoid the usual 'warning: variable xxx used but not defined' by declaring foreign (i.e. C) variables, using a *foreign declaration statement*. The syntax is simple:

```
declare xxx;
```

declares that `xxx` is defined somewhere outside the control of the SNL compiler. Foreign declarations may appear only at the top-level scope.

- Fixed the generated line markers, so that error and warning messages now correctly point to the source location (this was seriously broken in the old version).
- The syntax now accepts a larger subset of C. For instance, "character" literals are now recognized, as well as the `continue` statement.

1.5.2 Download

The project has been moved to

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

Releases can be downloaded from

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

A darcs repository containing the latest changes is located at

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

1.5.3 Build and Install

Apart from EPICS base (and its dependencies, e.g. Perl), building this version of seq requires an additional tool named re2c to be installed. This can be downloaded from http://sourceforge.net/projects/re2c/files/ (sources and Windows binaries), the home page is http://re2c.org/. If you are on a linux system, you will probably want to use the re2c package your distribution provides.

1.5.4 Internals

The compiler is not a re-write from scratch, but changes are numerous and pervasive.

The ancient versions of yacc and lexx that are bundled with EPICS base (in a modified version and thus never upgraded) are no longer used. Since lex/yacc suffers from severe backward compatibility disease (witness all the traditional-C stuff they still carry around, global vars and everything), I decided to look for something better. Shying away from more radical steps (for instance, it would have been much, much easier to re-implement the whole compiler in Haskell) because of all the usual issues involved (portability, nobody else would understand the code, etc, etc), I chose a more conservative approch: the new snc version uses re2c as the lexer generator, and lemon as the parser generator. Re2c is available for many platforms (including Windows), whereas lemon consists of just one source file (plus one template file) and so can be easily bundled with the sequencer. Both tools generate very fast code (reportedly better than lex/yacc).

Other internal changes include:

- use standard ANSI C
- clean separation between compiler stages: lexing, parsing, analysis, code generation
- no global variables, very few static ones
- unified error, warning, and debug reporting
- improved type safety by using unions instead of casts (plus a number of supporting macros) e.g. for the various syntactic constructs; added many new struct types
- use a hash table (the gpHash from libCom) for name lookup instead of doing linear search all over the place

- complete re-implementation of lexing and parsing (using re2c and lemon); the new parser spec has only three parsing conflicts and these are unavoidable: one is the well-known if-else ambiguity, the remaining two are due to escaped C code, where the parser cannot decide whether it is a declaration or a statement (the old version had a total of 744 conflicts)
- generated code contains fewer '#define's making accidental name clashes less probable
- the interface between the sequencer library and the generated code is now more type safe (no more XXX_FUNC casts, SS_ID and USER_VAR became anonymous struct types)
- in order to implement the state change command, an additional argument is needed for the action callback

1.6 Notes on Releases 2.0.0 to 2.0.12

1.6.1 Release 2.0.12

- Directory dependencies to permit building with the parallel make option -j.
- Library dependencies changes to match EPICS Base R3.14.10.
- Added the routine seqcaStats to src/seq/seq_qry.c.
- Cleaned up compiler warnings in src/snc.

1.6.2 Release 2.0.11

- A bugfix in src/seq/seq_ca.c found by Stephanie Allison and Till Straumann.
- Moved the PV build configuration variables from configure/RELEASE to the new CONFIG_SITE file, and moved CONFIG_APP functionality into CONFIG. Also set CHECK_RELEASE to YES by default in configure/Makefile.

1.6.3 Release 2.0.10

- Release 2.0.10 contains the new diagnostic seqcar(verbosity). This produces report about all the channel access connections from sequence programs.

1.6.4 Release 2.0.9

- seq_connect now initializes assignCount and numMonitoredChans BEFORE connecting to the PVs. With the previous versions it was possible to signal that everything was connected and the first monitor received before it actually happened.

1.6.5 Release 2.0.8

- seq_main.c was casting a pointer to an int. On some architectures this caused a warning message.
- Support for cygwin32 om windows was added.
- test/pv/ was still still using osiThread.h. This is replaced bt epicsThread.
- test/simple and test/validate now use EPICS_BASE_IOC_LIBS instead of EPICS_BASE_HOST_LIBS.

1.6.6 Release 2.0.7

- A sequence program could wait up to 20 seconds before all PVs connect and the first monitor occurs for each monitored PV. This could happen even if all PVs are local. This is now fixed.

1.6.7 Release 2.0.6

- When looking to see if all PVs have connected it now looks for first monitor coming back as well as all PVs connecting. This prevents the chance of using the value of a PV before it is given a value.

1.6.8 Release 2.0.5

- Replaced the C++ static constructor used for command registration with a generated registrar routine that must be listed in an IOC's xxxInclude.dbd file for use on on non-vxWorks systems. This approach permits sequence programs to be placed in an external support library and pulled in automatically by adding the registrar() statement in the xxxInclude.dbd and linking the IOC application against that library. For a sequence program that starts:

  ```
  program demo ...
  ```

  the dbd file should contain the statement:

  ```
  registrar(demoRegistrar)
  ```

  This is only required for applications that use the iocshell, vxWorks IOCs will work as before.
- Modifications to the demo/test programs needed to run these under Base R3.14.3.
- The config directory has been removed.
- Replaced devSequencer.c with version supplied by Kukhee Kim, SLAC

1.6.9 Release 2.0.4

Changes have been made to follow the R3.14.2 build rules. Generate an example application to see how to build sequencer applications. Note that this version requires R3.14.2 of base.

1.6.10 Release 2.0.1

Eric Norum, Janet Anderson, and I [Marty Kraimer] spent some time making the sequencer work with EPICS release 3.14.0alpha2. We are NOT responsible for the sequencer. Release 2.0.1 should be considered an interim release until Greg White and Ron Chestnut have time to become familiar with the changes we made.

We did not update the SNL/SEQ manual. The following briefly describes documentation changes needed for the 2.0.0 reference manual.

- New Compiler Options
  - +i Code will be generated to automatically register sequence functions and state programs. This is the default
  - -i Do not generate registration code. Only useful on vxWorks to save a small amount of memory.
- Compiling and linking a state program under Unix.
- The exampleApp supplied with base demonstrates how to build sequence programs that work with epics databases. It also shows how to build a sequence program that runs without databases.
- test/simple also shows how to build a sequence program that runs without databases running in the same process.
- The main program generated by the +m compiler options looks like:

```
int main(int argc,char *argv[]) {
    char * macro_def;
    epicsThreadId threadId;
    int callIocsh = 0;
    if(argc>1 && strcmp(argv[1],"-s")==0) {
        callIocsh=1;
        --argc; ++argv;
    }
    macro_def = (argc>1)?argv[1]:NULL;
    threadId = seq((void *)&snctest, macro_def, 0);
    if(callIocsh) {
        iocsh(0);
    } else {
        epicsThreadExitMain();
    }
    return(0);
}
```

Such a program is invoked via the call like:

```
<name> -s "xxx=xxx,..."
```

If the -s option is specified iocsh is started. This allows the user to issue sequence commands like seqShow. If the -s option is not given, the shell is not started. If a program is started as a background process the -s option should not be specified.

The other option argument is macro substitutions.

Comments about changes made for release 2.0.1

- I think config should be removed. Just require 3.14 and later.
- in test/simple and test/validate:

  ```
  assign v to "grw:xxxExample";
  ```

  changed to

  assign v to "{user}:xxxExample";
- in src/dev: * dev.c renamed to devSequencer.c * seq.dbd renamed to devSequencer.dbd
- in src/snc * The generated main program is different. See below * New option -i. See above * Made many changes to get rid of warning messages.
- in src/seq * seq now returns an epicsThreadId * all addesses now printed with %p * several epicsPrintf replaced with printf. * Got rid of simple shell supplied by seq. No longer needed.
- in src/pv
  - Only LIBRARY = pv is generated. It includes the stuff that was in pvKtl, pvCa, and pvFile if they are requested.
  - This was done because the old way caused a circircular dependency that is not accepted by windows.

1.7 Notes on Release 2.0

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.

1.7.1 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 *The PV (Process Variable) API*.

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.

1.7.2 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 *Global Entry and Exit Blocks*.

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 *State Entry and Exit Blocks*.

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*.

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 process variables associated with the elements of the array, which can be useful for parallel control. See *Queuing Monitors* and *syncQ*.

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 *Test* 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 Scope*.

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 *Asynchronous Use of pvPut* and *pvPutComplete*.

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* and *pvGet*.

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*.

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).

1.7.3 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.

1.7.4 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.

1.7.5 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.

Local variables These are partially supported. See Local variables.

pvNew dynamic loading This would remove some undesirable library dependencies. See *Overview*.

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.

1.8 Notes on Release 1.9

With version 1.9, 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 process variables.

Number of process variables The previous restriction on the number of process 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 process variables. This feature simplifies many codes that contain groups of similar variables. Furthermore, double-subscripted arrays allow arrays of array-valued variables.

Dynamic assignments Process 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 process 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

TWO

INSTALLATION

2.1 Prerequisites

You need to have an EPICS-3.14 base and its dependencies (GNU make, Perl) installed.

From 2.0.98 onwards, building the sequencer requires the lexer generator tool re2c. If you are on a Linux system, you will probably want to use the re2c package your distribution provides, otherwise sources and windows binaries are available from the re2c download page.

2.2 Download

Newer releases are available here:

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

The latest release is currently 2.0.99, which is a beta release for the upcoming 2.1.0. Release notes are contained in the *Introduction*.

Older releases can be either downloaded from the PSI website

http://epics.web.psi.ch/software/sequencer/download

or by following the links here:

Version	Release Notes
2.0.12	*Release 2.0.12*
2.0.11	*Release 2.0.11*
2.0.10	*Release 2.0.10*
2.0.9	*Release 2.0.9*
2.0.8	*Release 2.0.8*
2.0.7	*Release 2.0.7*
2.0.6	*Release 2.0.6*
2.0.5	*Release 2.0.5*
2.0.4	*Release 2.0.4*
2.0.3	n/a
2.0.2	n/a
2.0.1	*Release 2.0.1*
2.0.0	*Notes on Release 2.0*
1.9.5	*Notes on Release 1.9*
1.9.4	*Notes on Release 1.9*

You can look at (or even use at your own risk ;-) my latest development version by checking out the darcs repository:

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

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

2.3 Unpack

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

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

or, if you have GNU tar, simply:

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

You can now:

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

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

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

2.4 Configure and Build

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

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

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

Note: setting `INSTALL_LOCATION_APP` currently does not work.

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

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

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

2.5 Building the Manual

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

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

The html pages are generated by issuing:

```
> make docs=1
```

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

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

```
> make docs=1 pdf=1
```

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

2.6 Test

Under Linux and 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 `$SEQ/lib/$EPICS_HOST_ARCH`.

Change directory to test/demo and do:

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

to rune the demo program. This includes its own CA server, so no IOC or portable CA server is needed. The output should look something like this:

```
ben@sarun[1]: .../test/demo > ../../bin/linux-x86/demo stcmd.host
dbLoadDatabase "../../dbd/demo.dbd"
demo_registerRecordDeviceDriver(pdbbase)
dbLoadRecords "demo.db"
iocInit
Starting iocInit
############################################################################
###  EPICS IOC CORE built on Nov 17 2009
###  EPICS R3.14.8.2 $R3-14-8-2$ $2006/01/06 15:55:13$
############################################################################
iocInit: All initialization complete
seq &demo
SEQ Version 2.0.12: Thu May 13 22:53:48 2010
Spawning state program "demo", thread 0x8078348: "demo"
epics> demo_1 2010/05/13 23:17:15: start -> ramp_up
Spawning thread 0x8096660: "demo_1"
Spawning thread 0x809f330: "demo_2"
demo 2010/05/13 23:17:20: light_off -> light_on
demo_1 2010/05/13 23:17:25: ramp_up -> ramp_down
demo 2010/05/13 23:17:30: light_on -> light_off
demo_1 2010/05/13 23:17:35: ramp_down -> ramp_up
demo 2010/05/13 23:17:40: light_off -> light_on
demo_1 2010/05/13 23:17:45: ramp_up -> ramp_down
demo 2010/05/13 23:17:51: light_on -> light_off
demo_1 2010/05/13 23:17:55: ramp_down -> ramp_up
...
```

If you see the "start -> ramp_up" etc. messages, things are good. From another shell, do the following:

```
> caget ss0
ss0 light
> caget ss1
ss1 ramp
```

This illustrates the very basic "sequencer device support" that was added in release 2.0. These records are returning the names of the first two state-sets of the demo program.

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

```
ben@sarun[1]: .../bin/linux-x86 > ./sncExitOptx -s
SEQ Version 2.0.12: Thu May 13 22:53:48 2010
Spawning state program "sncexitoptx", thread 0x804ecf0: "sncexitoptx"
epics> 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'
...etc...
```

Note the *-s* argument to the test program. Without it, the program will not start an iocShell and you will have difficulty terminating the program (in fact, you have to `kill -9` it). Similar for the other test programs.

2.7 Use

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

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

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

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

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

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

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

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

2.8 Report Bugs

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

2.9 Contribute

I am always happy to receive patches (bug fixes, improvements, whatever). You can create a local copy of the darcs repository by saying:

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

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

```
darcs pull
```

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

```
darcs record
```

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

```
darcs send
```

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

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

CHAPTER

THREE

TUTORIAL

This chapter gives a gentle introduction to State Notation Language.

3.1 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 2-1. 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.

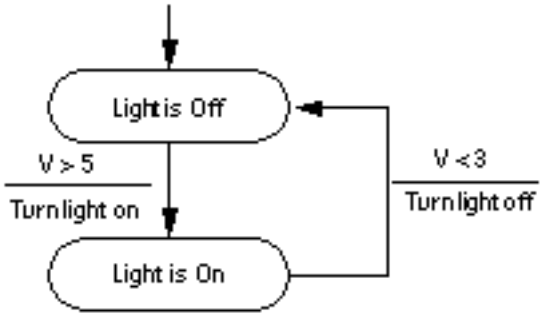

Figure 2-1: A simple state transition diagram

3.2 Elements of the State Notation Language

The following SNL code segment expresses the STD in The State Transition Diagram

```
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 process 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 process 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 process 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).

3.3 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 { ... }
```

3.4 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 following STD:

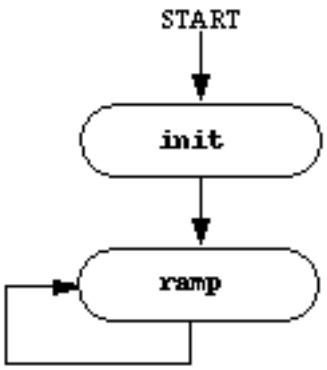

Figure 2-2: Structure of **generate_voltage** State Set

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 process 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.

3.5 Variable Names Using Macros

One of the features of the SNL and run-time sequencer is the ability to specify the names of process 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 process 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 *Specifying Run-Time Parameters* for more on macros.

3.6 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 process variable. The type declared does not have to be the same as the native control system value type. The conversion between types is performed at run-time.

You may specify array variables as follows:

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

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

3.7 Arrays of Variables

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

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

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

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

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

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

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

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

Alternatively, we can do this for the entire array:

```
monitor Vin;
```

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

Double subscripts offer additional options:

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

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

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

3.8 Dynamic Assignment

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

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

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

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

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

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

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

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

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

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

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

The following inequality will always hold:

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

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

3.9 Status of Process Variables

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

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

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

3.10 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 process variable that is being monitored. In that case, whenever a monitor is delivered, the corresponding event flag is set. Note that this provides an alternative to testing the value

of the monitored channel and is particularly valuable when the channel being tested is an array or when it can have multiple values and an action must occur for any change.

This example shows a state set that forces a low limit always to be less than or equal to a high limit. The first `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 process variable. If the SNL variable is an array then the event flag is set whenever a monitor is posted on any of the process variables that are associated with an element of that array.

3.11 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;

ss queue {
  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.

3.12 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 *Compiler Options*). The operation might not be initiated until the action statements in the current transition have been completed and it could complete at any later time. To test for completion use the function `pvGetComplete`, which is described in *Built-in Functions*.

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

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

3.13 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
```

3.14 Connection Management

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

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

These routines are described in *Built-in Functions*. If a variable disconnects or re-connects during execution of a 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 Synchronizing State Sets with Event Flags for an example.

3.15 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.

3.16 Process Variable Element Count

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

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

CHAPTER

FOUR

COMPILING SNL PROGRAMS

4.1 *snc*, the SNL to C Compiler

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

4.1.1 Compiler Options

SNC options start by a plus or minus sign, followed by a single character. A plus sign turns the option on, and a minus turns the option off.

Option	Description
+a	Asynchronous `pvGet`: the program continues without waiting for completion of the `pvGet` operation.
-a	Synchronous `pvGet`: the program waits for completion. This is the default if an option is not specified.
+c	Wait for process variables to connect before allowing the program to begin execution. This is the default.
-c	Allow the program to begin execution before connections are established to all channel.
+d	Turn on run-time debug messages.
-d	Turn off run-time debug messages. This is the default.
+e	Use the new event flag mode. This is the default.
-e	Use the old event flag mode (clear flags after executing a when statement).
+l	Add line markers to the generated code, so that C compiler messages refer to the SNL source file. This is the default.
-l	Do not produce line markers.
+m	Generate a main procedure for a stand-alone program.
-m	Do not generate a main procedure. This is the default.
+r	Make the generated code reentrant, thus allowing more than one instance of the state program to run on an IOC.
-r	Generated code is not reentrant. This is the default.
+w	Display SNC warning messages. This is the default.
-w	Suppress SNC warnings.

Note that both *+a* and *-a* options are ignored for calls to `pvGet` that explicitly specify `SYNC` or `ASYNC` (in the 3rd argument, see *pvGet*).

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

4.1.2 Output File

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

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

4.1.3 Errors

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

4.1.4 Warnings

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

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

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

4.2 C Pre-processor

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

4.3 Complete Build

The C code generated by *snc* from an SNL program is not a complete program, but merely a collection of procedures, data types, and variables. The generated procedures are supposed to be called by the sequencer library, which must be linked to the program. Furthermore, the generated code includes a number of header files, both from the sequencer and from EPICS base. Thus the compiler needs to have the EPICS base and sequencer include directories in its include path, and when linking it needs to link with sequencer and some EPICS base libraries.

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

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

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

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

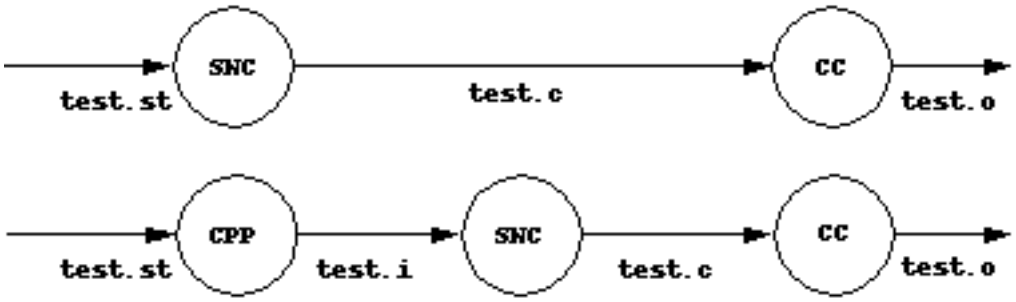

Figure 3-1: Two Methods of Compiling a State Program

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

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

where `<osclass>` corresponds to your operating system, see Section 4.3.6 "Specifying osclass specific definitions" of the EPICS Application Developer's Guide. Here is an excerpt:
For vxWorks-* targets <osclass> is vxWorks.
For RTEMS-* targets <osclass> is RTEMS.
For solaris-* targets <osclass> is solaris.
For win32-* targets <osclass> is WIN32.
For linux-* targets <osclass> is Linux.
For hpux-* targets <osclass> is hpux.
For darwin-* targets <osclass> is Darwin.
For aix-* targets <osclass> is AIX.

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

4.4 Building a Stand-alone Program for 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.

4.5 Using makeBaseApp

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

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

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

All that's left to do is add:

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

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

CHAPTER

FIVE

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.

5.1 VxWorks-specific instructions

5.1.1 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
```

5.1.2 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.

5.1.3 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.

5.1.4 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 process 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
```

5.1.5 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.

5.2 Unix-specific instructions

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.3 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 (see *Program*). Parameters specified on the command-line override those specified in the `program` statement.

5.3.1 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"
```

5.3.2 Unix

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

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

5.3.3 Access within program

Parameters can be accessed by your program with the function *macValueGet*. 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).

5.4 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.

5.5 What Triggers an Event?

There are five types of sequencer event:

- a process variable monitor is posted
- an asynchronous `pvGet` or `pvPut` completes
- a time delay elapses
- an event flag is set or cleared
- a process 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

SIX

SNL REFERENCE

This chapter is supposed to give a detailed reference for the SNL syntax and semantics.

Formal syntax is given in BNF. Multiple rules for the same *nonterminal symbol* mean that any of the given rules may apply.

6.1 Lexical Syntax

6.1.1 Comments

```
comment  ::=   "/*" <anything> "*/"
```

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

6.1.2 Identifiers

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

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

6.1.3 Literals

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

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

Note that array and struct initializers are not supported.

6.1.4 Embedded C Code

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

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

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

See Escape to C Code for examples and rationale.

Embedded C code fragments are causing two of the three conflicts in the grammar. The reason is that the parser cannot decide whether such a fragment is a declaration or a statement.

6.1.5 Line Markers

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

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

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

Line markers are typically generated by preprocessors, such as CPP.

6.2 Program

```
program  ::=   "program" identifier program_param global_defns entry state_sets exit c_c
```

This is the overall structure of an SNL program. After the keyword "program" comes the name of the program, followed by an optional program parameter, global definitions, an optional entry block, the state sets, an optional exit block, and finally some embedded c code.

6.2.1 Program Name and Parameter

The program name is an identifier. It 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 (see *Specifying Run-Time Parameters*).

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

The program name may be followed by an optional string enclosed in parentheses. The string content must be a list of comma-separated parameters in the same form as they are specified on the command line (see *Specifying Run-Time Parameters*). Command-line parameters override those specified here.

```
global_defns  ::=  global_defns global_defn
global_defns  ::=  global_defn
global_defn   ::=  assign
global_defn   ::=  monitor
global_defn   ::=  sync
global_defn   ::=  syncq
global_defn   ::=  decl
global_defn   ::=  evflag_decl
global_defn   ::=  foreign_decl
global_defn   ::=  option
global_defn   ::=  c_code
```

Global (top-level) definitions, see Definitions for details.

6.2.2 Global Entry and Exit Blocks

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

A state program may specify optional entry code to run prior to state set thread creation, and exit code to run prior to thread deletion. Both are run in the context of the first state set thread, before the other threads are created, resp. after they have been deleted. The entry or exit code is a regular SNL code block an thus can contain local definitions. However, no process variable access functions may be called within the entry code.

Global entry and exit blocks should not be confused with the entry and exit blocks of a state, which have the same syntax, but are executed at each transition from/to a new state.

6.2.3 Final C Code Block

```
c_codes  ::=   c_codes c_code
c_codes  ::=
c_code   ::=   embedded_c_code
```

At the end of the program may come any number of embedded C code blocks. See Escape to C Code.

6.3 Definitions

6.3.1 Declarations

Scalars

```
decl  ::=   type identifier ";"
decl  ::=   type identifier "=" expr ";"
type  ::=   "char"
type  ::=   "short"
type  ::=   "int"
type  ::=   "long"
type  ::=   "unsigned" "char"
type  ::=   "unsigned" "short"
type  ::=   "unsigned" "int"
type  ::=   "unsigned" "long"
type  ::=   "float"
type  ::=   "double"
type  ::=   "string"
```

Variable declarations are similar to C but limited in a number of ways:

- only integers, floating point numbers are allowed as base types
- only scalar initialization is permitted
- only one variable may be declared per declaration

The type `string` can be viewed as if it were defined in C as:

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

where `MAX_STRING_SIZE` is supposed to be defined in one of the included header files form EPICS base. The exact value depends on the EPICS base version (but in all versions is at least 40).

Arrays and Pointers

```
decl  ::=  type identifier subscript ";"
decl  ::=  type identifier subscript subscript ";"
decl  ::=  type "*" identifier ";"
decl  ::=  type "*" identifier subscript ";"
```

SNL allows declaration of one- or two-dimensional arrays of scalar elements, as well as pointers to scalars, and (one-dimensional) arrays of such pointers.

Note that arrays of strings and event flags are not supported.

Event Flags

```
evflag_decl  ::=  "evflag" identifier ";"
```

This declares an event flag, see *Synchronizing State Sets with Event Flags*.

Foreign Declarations

```
foreign_decl  ::=  "declare" identifier ";"
```

This declares the named C variable or CPP macro, so the SNL compiler knows about it. No warning will be issued if such a variable (or macro) is used in the program.

6.3.2 Variable Scope

Global variables, that is, variables declared at the top-level (before any state sets) are visible throughout the program, and persist as long as the program is running. We say they have *program life time*.

Local variable declarations come in two flavours, depending on where they appear:

1. Variables of *program life time* are global variables and those which are local to a state set or a state clause. Only variables of this sort can be assigned to a process variable, monitored, synced etc.
2. Variables declared in any other block have lifetime *limited to the enclosing block*, they disappear when the block exits, just as block local variables in C. They can *not* be assigned to process variables.

6.3.3 Process Variables

assign

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

This assigns program variables to named process variables.

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

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

The process variable name may contain macro names enclosed in braces, as in `"{sys}{sub}voltage"`. Macros are named following the same rules as C language variables. Macros are defined via command line argument, or program parameter (see Program Name and Parameter). If process variable name is an empty string, then no actual assignment is performed, but the variable is marked for potential (dynamic) assignment with `pvAssign`.

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

Pointer types may not be assigned to process variables.

monitor

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

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

Monitored variables are automatically updated with the current value of the underlying process variable.

sync

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

An event flag can be associated with an SNL variable (which may be an array, and thus associated with several process variables). When a monitor is posted on any of the associated process variables, the corresponding event flag is set (even if it was already set).

syncQ

```
syncq       ::=   "syncQ" identifier opt_subscript to identifier syncq_size ";"
syncq_size  ::=   integer_literal
syncq_size  ::=
```

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 process 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 process 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.

6.3.4 Option

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

Option values must include the "+" or "-" sign. Example:

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

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

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

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

6.4 State Set

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

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

State set names must be unique in the program.

6.4.1 State Set Local Definition

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

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

See Variable Scope for details on what local definitions mean.

6.4.2 State

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

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

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

State Local Definition

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

State Option

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

The state options are:

+t Reset delay timers each time the state is entered, even if entered from the same state.

-t

Don't reset delay timers when entering from the same state. In other words, the *delay* function will return whether the specified time has elapsed from the moment the current state was entered from a different state, rather than from when it was entered for the current iteration.

+e Execute *entry* blocks only if the previous state was not the same as the current state. This is the default.

-e Execute *entry* blocks even if the previous state was the same as the current state.

+x Execute *exit* blocks only if the next state is not the same as the current state. This is the default.

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

For example:

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

State Entry and Exit Blocks

The syntax is the same as for global entry blocks:

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

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

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

Note that state options can be used to control whether entry/exit blocks get executed even if the new state is the same as the current one.

Transitions

```
whens     ::=   whens when
whens     ::=   when
when      ::=   "when" "(" opt_expr ")" block "state" identifier
opt_expr  ::=   expr
opt_expr  ::=
```

A state transition starts with the keyword "when", followed by a condition (in parantheses), followed by a block, and finally the name of the target state (which must be any state of the same state set).

If there is no condition given, it defaults to `TRUE` (1).

6.4.3 Block

```
block        ::=   "{" block_defns statements "}"
block_defns  ::=   block_defns block_defn
block_defns  ::=
block_defn   ::=   decl
block_defn   ::=   c_code
```

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

Block definitions are: declarations and embedded C code.

6.5 Statements and Expressions

6.5.1 Statements

```
statements     ::=   statements statement
statements     ::=
statement      ::=   "break" ";"
statement      ::=   "continue" ";"
statement      ::=   "state" identifier ";"
statement      ::=   c_code
statement      ::=   block
statement      ::=   "if" "(" expr ")" statement
statement      ::=   "if" "(" expr ")" statement "else" statement
statement      ::=   "while" "(" expr ")" statement
statement      ::=   for_statement
statement      ::=   opt_expr ";"
for_statement  ::=   "for" "(" exprs ";" opt_expr ";" exprs ")" statement
```

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

6.5.2 Expressions

Formation rules for expressions are listed in groups of decending order of precendence.

Atomic Expression

```
expr    ::=   integer_literal
expr    ::=   floating_point_literal
expr    ::=   string
expr    ::=   identifier
string  ::=   string_literal
```

These are literals and variables.

Parenthesized Expression

```
expr  ::=   "(" expr ")"
```

Primary Expression Operators

```
expr  ::=   identifier "(" exprs ")"
expr  ::=   "exit" "(" exprs ")"
expr  ::=   "delay" "(" exprs ")"
expr  ::=   expr "[" expr "]"
expr  ::=   expr "."  expr
expr  ::=   expr "->" expr
expr  ::=   expr "++"
expr  ::=   expr "-"
```

These are: function call, array subscript, record selection, pointer to record selection, and postfix operators (increment and decrement).

Note that `exit` and `delay` are listed explicitly because they are keywords, not identifiers, but can also be used as functions.

Note also that SNL makes no use of the semantics of structure member access (a side-effect that you may notice is that the state notation compiler will warn that structure tags are unused variables).

Unary Prefix Operators

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

Left-associative Binary Operators

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

Ternary Operator

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

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

Assignment Operators

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

These operators are right-associative.

Expression List

```
exprs  ::=   exprs "," expr
exprs  ::=   expr
exprs  ::=
```

Comma separated expression lists are not expressions (in SNL), since SNL does not support the comma operator. However, they appear as parts of expressions and statements, and then have the same meaning as in C.

6.6 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 *assigned_var* refers to any SNL variable that is assigned to a process 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 process 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.

6.6.1 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 (see State Option) controls whether the delay is measured from when the current state was entered from a different state (`-t`) or from any state, including itself (`+t` , the default)

6.6.2 pvPut

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

This function puts (or writes) the value of an SNL variable to the underlying process 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.

6.6.3 pvPutComplete

`int` **`pvPutComplete`**(*assigned_var*)
`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 process 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 process 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.

6.6.4 pvGet

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

This function gets (or reads) the value of an SNL variable from the underlying process 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.

6.6.5 pvGetComplete

`int` **`pvGetComplete`**(*assigned_var*)

This function returns *TRUE* if the last get of this process 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.

6.6.6 pvGetQ

`int` **`pvGetQ`**(*assigned_var*)
`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.

6.6.7 pvFreeQ

`void` **`pvFreeQ`**(*assigned_var*)

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.

6.6.8 pvMonitor

`int` **`pvMonitor`**(*assigned_var*)

This function initiates a monitor on the underlying process variable.

6.6.9 pvStopMonitor

`int` **`pvStopMonitor`**(*assigned_var*)

This function terminates a monitor on the underlying process variable.

6.6.10 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).

6.6.11 pvCount

`int` **`pvCount`**(*assigned_var*)

This function returns the element count associated with the process variable.

6.6.12 pvStatus

`pvStat` **`pvStatus`**(*assigned_var*)

This function returns the current alarm status for the process 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.

6.6.13 pvSeverity

`pvSevr` **`pvSeverity`**(*assigned_var*)

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

6.6.14 pvTimeStamp

`TS_STAMP` **`pvTimeStamp`**(*assigned_var*)

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.

6.6.15 pvAssign

`int` **`pvAssign`**(*var, process_variable_name*)

This function assigns or re-assigned the SNL variable `var` to `process_variable_name`. If `process_variable_name` is an empty string then `var` is de-assigned (not associated with any process variable).

As usual, `var` can also be an array element.

Note: `pvAssign` can only be called on variables (or array elements) that have been statically marked as process variables using the **`assign`** syntax. An empty string may be used for the initial assignment.

A better name for this function would be `pvReassign`.

6.6.16 pvAssigned

`int` **`pvAssigned`**(*assigned_var*)

This function returns *TRUE* if the SNL variable is currently assigned to a process variable.

6.6.17 pvConnected

`int` **`pvConnected`**(*assigned_var*)

This function returns *TRUE* if the underlying process variable is currently connected.

6.6.18 pvIndex

`int` **`pvIndex`**(*assigned_var*)

This function returns the index associated with a process variable. See User Functions within the State Program.

6.6.19 pvChannelCount

`int` **`pvChannelCount`**()

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

6.6.20 pvAssignCount

`int` **`pvAssignCount`**()

This function returns the total number of SNL variables in this program that are assigned to underlying process 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*.

6.6.21 pvConnectCount

`int` **`pvConnectCount`**`()`

This function returns the total number of underlying process variables that are connected. Note: if all variables are connected then the following expression is *TRUE*:

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

6.6.22 efSet

`void` **`efSet`**`(`*event_flag*`)`

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.

6.6.23 efTest

`int` **`efTest`**`(`*event_flag*`)`

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

6.6.24 efClear

`int` **`efClear`**`(`*event_flag*`)`

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.

6.6.25 efTestAndClear

`int` **`efTestAndClear`**`(`*event_flag*`)`

This function clears the event flag and returns *TRUE* if the event flag was set. It is intended for use within a *when* clause.

6.6.26 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*.

6.7 C Compatibility Features

6.7.1 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, 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.

6.7.2 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.

6.7.3 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 process 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 (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.

6.7.4 Variable Modification for Reentrant Option

If the reentrant option (*+r*) is specified to SNC then all variables are made part of a structure. Suppose we have the following 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.

6.8 Syntax Summary

```
program        ::=  "program" identifier program_param global_defns entry state_sets ex
program_param  ::=  "(" string ")"
program_param  ::=
global_defns   ::=  global_defns global_defn
global_defns   ::=  global_defn
global_defn    ::=  assign
global_defn    ::=  monitor
global_defn    ::=  sync
global_defn    ::=  syncq
global_defn    ::=  decl
global_defn    ::=  evflag_decl
global_defn    ::=  foreign_decl
global_defn    ::=  option
global_defn    ::=  c_code
assign         ::=  "assign" identifier to string ";"
assign         ::=  "assign" identifier subscript to string ";"
assign         ::=  "assign" identifier to "{" strings "}" ";"
strings        ::=  strings "," string
strings        ::=  string
monitor        ::=  "monitor" identifier opt_subscript ";"
sync           ::=  "sync" identifier opt_subscript to identifier ";"
syncq          ::=  "syncQ" identifier opt_subscript to identifier syncq_size ";"
syncq_size     ::=  integer_literal
syncq_size     ::=
to             ::=  "to"
to             ::=
opt_subscript  ::=  subscript
opt_subscript  ::=
subscript      ::=  "[" integer_literal "]"
foreign_decl   ::=  "declare" identifier ";"
evflag_decl    ::=  "evflag" identifier ";"
decl           ::=  type identifier ";"
decl           ::=  type identifier "=" expr ";"
decl           ::=  type identifier subscript ";"
decl           ::=  type identifier subscript subscript ";"
decl           ::=  type "*" identifier ";"
decl           ::=  type "*" identifier subscript ";"
type           ::=  "char"
type           ::=  "short"
type           ::=  "int"
type           ::=  "long"
type           ::=  "unsigned" "char"
type           ::=  "unsigned" "short"
type           ::=  "unsigned" "int"
type           ::=  "unsigned" "long"
type           ::=  "float"
type           ::=  "double"
type           ::=  "string"
option         ::=  "option" option_value identifier ";"
option_value   ::=  "+"
option_value   ::=  "-"
state_sets     ::=  state_sets state_set
state_sets     ::=  state_set
state_set      ::=  "ss" identifier "{" ss_defns states "}"
ss_defns       ::=  ss_defns ss_defn
ss_defns       ::=
ss_defn        ::=  assign
ss_defn        ::=  monitor
ss_defn        ::=  sync
ss_defn        ::=  syncq
ss_defn        ::=  decl
states         ::=  states state
```

CHAPTER

SEVEN

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.

7.1 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.

7.2 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.

7.3 A tour of the API

7.3.1 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).

7.3.2 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;
}
```

7.3.3 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;
}
```

7.4 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.

7.4.1 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 process 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 process variable has fewer than `count` elements
- `status` comes from the underlying message system
- it is converted to a `pvStat`

7.4.2 Class pvSystem

`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*. 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 wait for events from multiple `pvSystem`.

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.

7.4.3 Class pvVariable

`pvVariable` is an abstract class that must be extended by specific message systems. It corresponds to a process 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 process 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` objects to be associated with the same underlying process 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` .

7.5 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.

7.5.1 Check-list

This section gives a check-list. See Example 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 for an example.

Edit configure/RELEASE

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

7.5.2 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 A tour of the API. 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

EIGHT

EXAMPLES OF STATE PROGRAMS

8.1 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");
      printf("Another thing to do on entry");
    }
    when (v>5.0) {
      printf("now changing to high\n");
    } state high
    when (delay(.1)) { } state low
    exit {
      printf("Something to do on exit");
    }
  }

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

8.2 Dynamic assignment example

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

8.3 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 *A Complete State Program*. It generates a triangle waveform in one state set and detects the level in another. Other state sets detect and print alarm status and demonstrate asynchronous `pvGet` and `pvPut` operation. The program demonstrates several other concepts, including access to run-time parameters with macro substitution and `macValueGet` , use of arrays, escaped C code, and VxWorks input-output.

8.3.1 Preamble

```
/* 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 */
```

8.3.2 level_det state set

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

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

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

8.3.3 generate_voltage state set

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

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

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

8.3.4 test_status state set

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

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

```
      prev_status = ch_status;
    } state status_check
  }
}
```

8.3.5 periodic_read state set

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

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

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

8.3.6 exit procedure

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

8.3.7 C functions

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

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

```
    char *pstr;

    switch (status)
    {
      case NO_ALARM:    pstr = "no alarm";        break;
      case HIHI_ALARM:  pstr = "high-high alarm"; break;
      case HIGH_ALARM:  pstr = "high alarm";      break;
      case LOLO_ALARM:  pstr = "low-low alarm";   break;
      case LOW_ALARM:   pstr = "low alarm";       break;
      case STATE_ALARM: pstr = "state alarm";     break;
      case COS_ALARM:   pstr = "cos alarm";       break;
      case READ_ALARM:  pstr = "read alarm";      break;
      case WRITE_ALARM: pstr = "write alarm";     break;
      default:          pstr = "other alarm";     break;
    }
    fprintf (fd, "Alarm condition: \"%s\"", pstr);
    if (severity == MINOR_ALARM)
      pstr = "minor";
    else if (severity == MAJOR_ALARM)
      pstr = "major";
    else
      pstr = "none";
    fdprintf (fd, ", severity: \"%s\", value=%g\n", pstr, value);
  }
}%
```

- *Index*

INDEX