Contents

- Brief overview of EPICS SNL (state notation language) and how it is used[1] [2] [3]
- Extent to which the EPICS tool is specific to EPICS
- Implications of port to Unix / CDEV environment
- EPICS sequencer wish list

1. There is some overlap with the sequencer presentation given at the collaborators' meeting last week
2. Some material has been borrowed from Andy Kozubal and Bob Dalesio's SNC training slides
3. Some material has been borrowed from Ned Arnold's state notation language overview slides

97/05/20
WILLIAM LUPTON
WLUPTON@KECK.HAWAII.EDU
SOFTWARE SHARING WORKSHOP

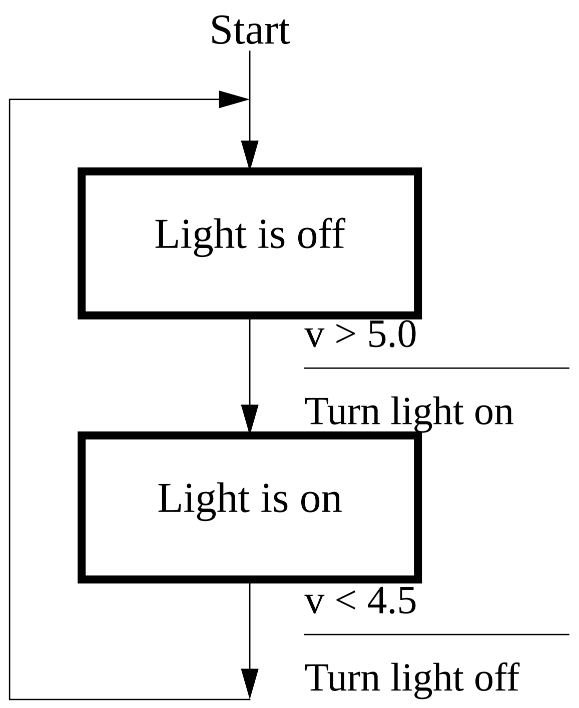
Overview (1)

- EPICS SNL is a language designed specifically for translating state transition diagrams into C code:
  - based on Mealy machine: actions are associated with events rather than with states
  - C-like: state notation compiler (SNC) converts SNL source code to C, which is then compiled
  - C code can be directly embedded (and external functions can be called)
  - multiple parallel diagrams can be implemented; event flags or variables allow communication
  - EPICS CA (channel access) is directly integrated: channels appear as module-local variables
  - “just a CA client”; can *theoretically* run on IOC or under Unix
  - implemented by Andy Kozubal (LANL) to run under VxWorks (major upgrades with v1.9)
  - port to Unix (XMSEQ) by Ben-chin Cha (ANL) is no longer supported

- Typical uses:
  - coordination of subsystems (automated startup or shutdown, complex closed loop control)
  - enforcing prudent operational procedures (gateway between user and low-level system)
  - fault detection (for complex fault modes which are not tied to single variables)
  - fault recovery (transition to safe state on detection of fault)
  - access to Unix file-system on IOC (save / restore)

Overview (2)

- The standard example (light on above 5.0v and off below 4.5v):



```
program lightDemo

double v;    /* module-local variable */
assign v to “demo:volts”;
monitor v;   /* “demo:volts” is CA name */

long light;
assign light to “demo:light”;

ss main {    /* “ss” means “state set” */

    state lightIsOff {
        when ( v > 5.0 ) {
            light = TRUE; pvPut( light );
        } state lightIsOn
    }

    state lightIsOn {
        when ( v < 4.5 ) {
            light = FALSE; pvPut( light );
        } state lightIsOff
    }
}
```

97/05/20
WILLIAM LUPTON
WLUPTON@KECK.HAWAII.EDU
SOFTWARE SHARING WORKSHOP

Overview (3)

- More about `assign` and `monitor`:
  - `assign` makes an association between a variable and a named EPICS channel
  - this association can be dynamic; you can change channel names at run-time
  - `monitor` arranges for the variable always to have the current value of the EPICS channel
  - event flags can be used to achieve finer control (*e.g.* wake-up on change; monitor queuing)
- More about `when()`:
  - states can have multiple `when()` clauses; the body of the first one that evaluates `TRUE` is executed
  - on entry to a state, when() conditions are re-evaluated; if none are `TRUE`, the sequence pends on an event (no polling)
- More about state sets:
  - a sequencer program may contain multiple state sets, each one of which is an independent thread of control and corresponds to a single state transition diagram
  - all state sets in a program see the same module-local variables; variables or event flags (not shown in the example) can be used to implement control flows
- Other notes:
  - run-time macro expansion permits parametrization of channel names etc. (can run multiple copies of same sequence, each with own set of variables)
  - arrays of channels can be used; useful for operating on sets of related channels

How specific to EPICS is the state machine?

- EPICS state machine concepts:
  - ❑ as is illustrated by the (admittedly simple) example, the state machine is not specific to EPICS
  - ❑ this example could work in any environment where objects are identified by name and where there is the concept of a monitor (admittedly, other systems might not call them monitors)
  - ❑ the only vaguely EPICS terms in the example are `monitor` and `pvPut` (the acronym PV is often found in EPICS and stands for the fairly neutral “process variable”)
  - ❑ in my experience, the same is also true of more complex examples: one doesn’t need to use EPICS-specific or OS-specific terms or routines in order to write EPICS sequences
  - ❑ perhaps this is not surprising, considering that until recently a Unix version of the sequencer was supported (using, I believe, the Florida State University Posix threads implementation)

- state notation compiler
  - ❑ the compiler uses `lex` and `yacc` and is written in ANSI C; it runs on many Unix systems
  - ❑ the generated ANSI C code contains only two references to EPICS or VxWorks (an `OPT_VXWORKS` option appears unused; the EPICS time-stamp format seems to be assumed)

- run-time environment
  - ❑ clearly, the run-time environment is a different matter and is bound to be heavily OS-specific; I haven’t looked closely to see how much run-time code is portable
  - ❑ to put this in context, the run-time code consists of a total of 3384 lines of C code (including comments) in seven modules; there are a total of 1245 semicolons

Implications of a port to Unix / CDEV

- I should declare that I have only a passing knowledge of CDEV and have not used it
- Threads:
  - ❑ most real sequences are inherently multi-threaded; under VxWorks, each state set is a separate task
  - ❑ state sets share a common address space, so a threaded solution is necessary under Unix
  - ❑ the underlying message system (CDEV and any services that it may activate) needs to work properly in this threaded environment
  - ❑ these problems must already have been solved for the ANL Unix sequencer implementation?
- Other:
  - ❑ C dependence: the EPICS sequencer allows inline C code and direct invocation of C routines; if this is to be retained, pre-processing into C (or maybe C++) must be retained
  - ❑ does this mean that use of Java and / or Java threads to provide the run-time environment is out of the question?

Wish list

- This list is slightly abbreviated from the one presented last week.
- Minor additions:
  - ❑ Allow action statements to be executed on entry to and on exit from a state
  - ❑ Support channel access put with callback (`pvPutAck()`)
  - ❑ Support more of the C language (initialization, ternary operator and local variables would be nice!)
  - ❑ Allow an action not to reset timers
- Major additions:
  - ❑ Optionally permit several state sets to share a single task
  - ❑ Permit a “subroutine” state, callable as an action *[note: pre-processor macros provide a partial solution to this]*
  - ❑ Permit (parametrized) hierarchical states that can be referenced in several places *[note: also can be partially addressed by pre-processing]*
  - ❑ Provide hooks for external C code to set and test event flags (including from ISRs)
  - ❑ Provide a comprehensive test suite
- Longer-term additions:
  - ❑ Integrate with display managers to provide direct interaction with display elements
  - ❑ Allow sequencer dynamically to create internal CA database
  - ❑ Define SNL in terms of C++ objects (or Java?)