On-line State Manager State Transition Models and Client API for STAR System Test

D. Olson
Submitted: 6-Sept-1996
Revised: 12-Sept-1996

This document describes the state transition models and the programming interface used by client subsystems of the on-line state manager used for the STAR system test. The prototype state manager described in this document exists in the onl/psm software package in the STAR software library. Additional documentation is available at http://www.rhic.bnl.gov/star/starlib/doc/www/html/ssd_l/psm_l/psm.html.

State Transition Models

There are three state transition models used by the prototype state manager, a sequencing state model, a command state model and an alarm state model. The sequencing state model consists of the states and transitions (events) which comprise the major control sequencing necessary for each subsystem (trigger, daq, hardware control components, monitoring components, etc). The command state model provides for an asynchronous command interface between a subsystem and the operator, independent of the sequencing state model. The alarm state model provides a synchronous messaging interface between the operator and the subsystem which is intended to be used for abnormal or error conditions.

Sequencing State Model

The state transition diagram for the sequencing state model is shown in figure 1. States are boxes, events are arrows which show transitions from one state to another, labels on arrows are the names of events, unlabelled arrows indicate automatic transitions that occur when the activity of one state has concluded successfully. The initial state is "Connected" and the final state is "Disconnected".

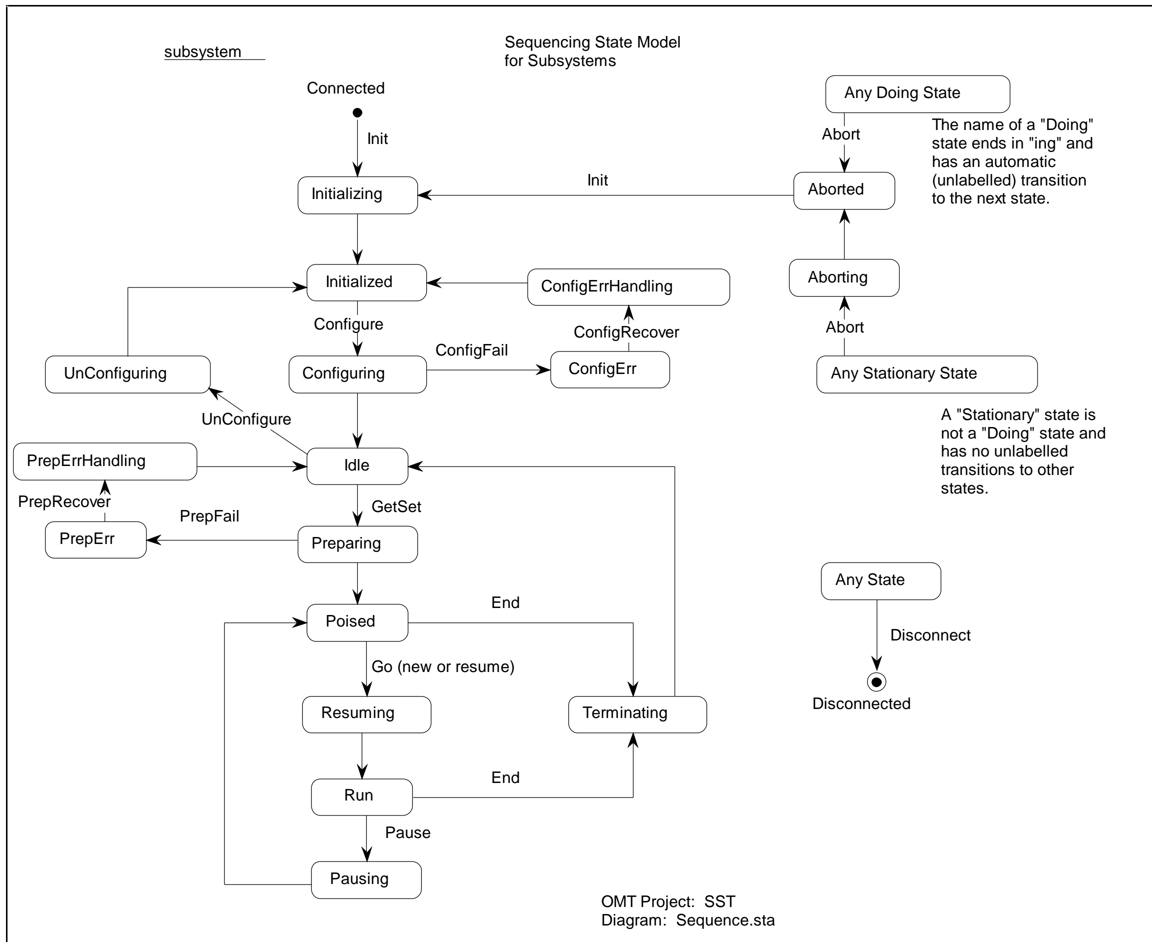

Figure 1. Sequencing state transition diagram.

Description of sequencing model states.

Aborted

The stationary state following an abort. The operator should be able to investigate conditions in the sub-system and try to diagnose the failure.

Aborting

A transitional state following an abort. The sub-system should preserve, as much as possible, the environment which caused the abort so that it may be investigated by the operator while in the Aborted state.

Connected

Initial stationary state.

ConfigErr

The stationary state entered following a configuring failure.

ConfigErrHandling

The transitional state in which the subsystem attempts to recover from the configuring error. If successful the subsystem will go to the Initialized state. If unsuccessful the subsystem should abort.

Configuring
: The transitional state in which a subsystem loads all of its external configuration data.

Disconnected
: The final sequencing state. In this state the subsystem no longer exists from the point of view of the state manager.

Idle
: The stationary state following configuring in which the subsystem has successfully loaded all of its configuration data.

Initialized
: The stationary state following initializing.

Initializing
: The transitional state in which the subsystem initializes its internal conditions.

Pausing
: The transitional state in which the subsystem attempts to suspend its “run” activity.

Poised
: The stationary state in which a subsystem is fully configured and any inter-subsystem interfaces used in the run state are ready.

Preparing
: The transitional state in which a subsystem gets ready to run. Inter-subsystem interfaces should be available for testing or communications during this state.

PrepErr
: The stationary state entered by a subsystem after it detects an error during the preparing state.

PrepErrHandling
: The transitional state during which the subsystem attempts to recover from the preparing error and return to idle.

Resuming
: The transitional state in which the subsystem is attempting to run.

Run
: The stationary state in which the subsystem is fully performing its primary function for operating the detector and taking data.

Terminating
: The transitional state in which the subsystem attempts to end its run activity.

UnConfiguring
: The transitional state in which the subsystem does an necessary activity so that it can return to the Initialized state. This will likely be followed by configuring with a new set of configuration data.

Command State Model

The command state model is shown in figure 2. States are shown as boxes. The initial state is “Ready” and the final state is “Disconnected.” The events, shown as arrows, cause transitions between states. The content of the command sent from the operator to the subsystem is the argument of the Go event.

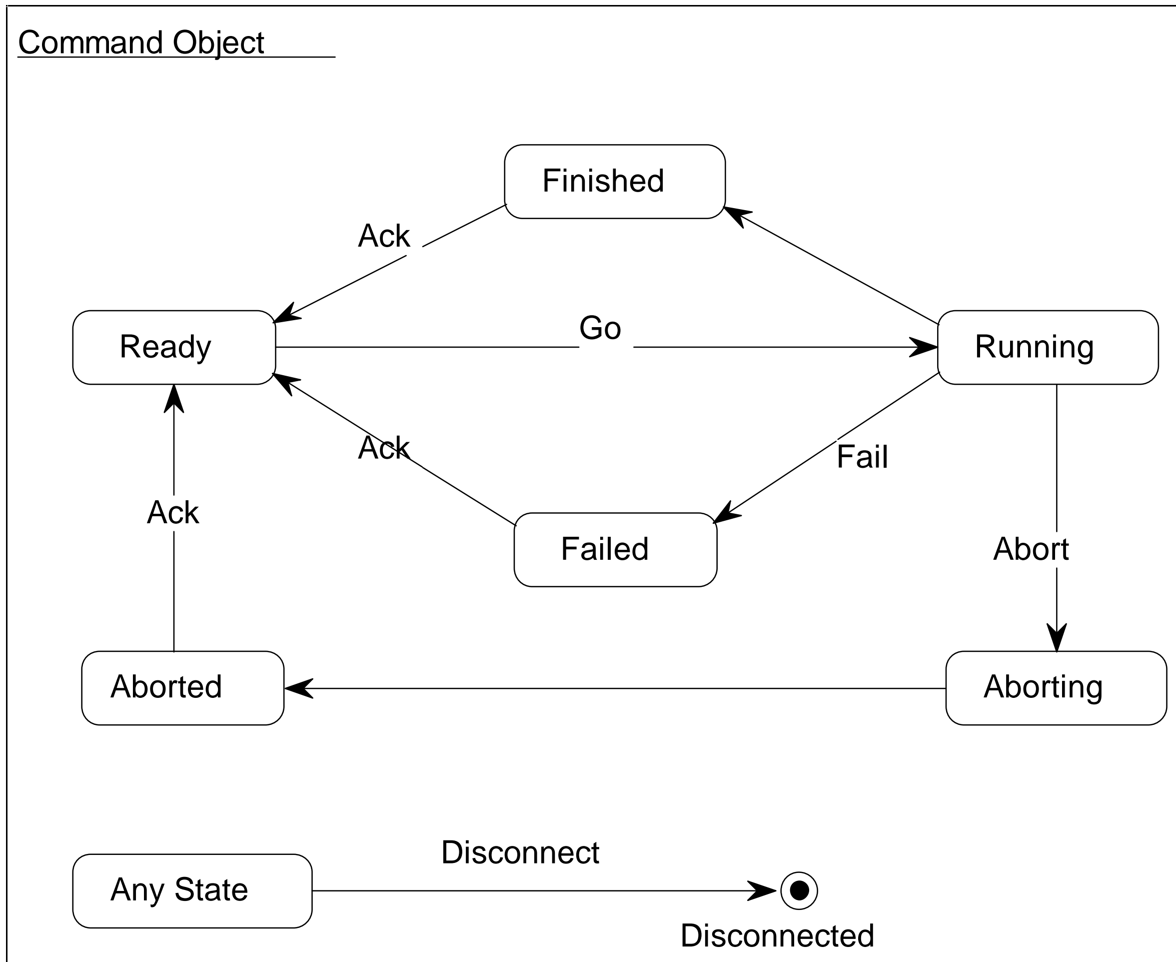

Figure 2. Command state transition diagram.

Command model states

Aborted
: The stationary state following an abort. The operator should be able to investigate conditions in the sub-system command handler and try to diagnose the failure.

Aborting
: A transitional state following an abort. The sub-system command handler should preserve, as much as possible, the environment which caused the abort so that it may be investigated by the operator while in the Aborted state.

Disconnected
: The final state following a disconnect event. In this state the command handler no longer exists to the state manager.

Failed
: The stationary state following a failure of the command to complete successfully.

Finished
: The stationary state following successful completion of the command.

Ready

The initial state and the stationary state in which the subsystem command handler is waiting for a command.

Running

The transitional state in which the command is executing.

Alarm State Model

The alarm state transition diagram is shown in figure 3. States are shown as boxes. The initial state is “Posted” and the final state is “Disconnected.” The events, shown as arrows, cause transitions between states. The content of the alarm message sent to the operator is the logmsg argument of smNewObject. The reply from the operator is the argument of the Ack event.

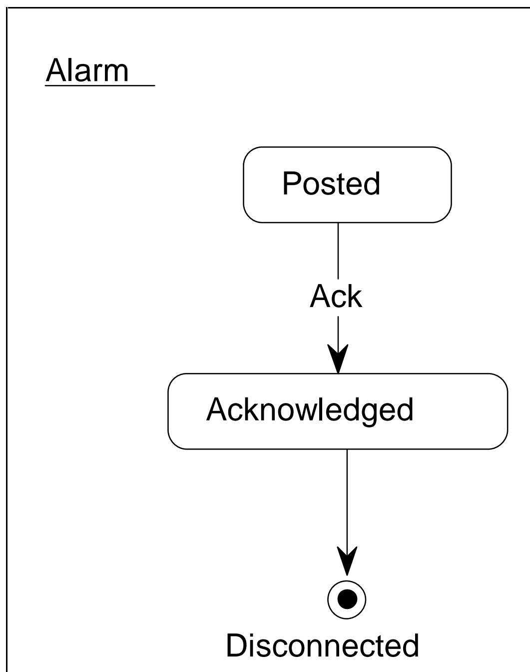

Figure 3. Alarm model state transition diagram.

Alarm model states

Acknowledged

The transitional state in which the subsystem deals with the operator’s response to the alarm. The operator’s response is received as the argument to the Ack event.

Disconnected

The final state. In this state the command handler no longer exists to the state manager.

Posted

The initial stationary state in which the subsystem alarm object waits for an operator response.

State Manager Client API

This section documents the programming interface for clients of the prototype state manager in the onl/psm package in the STAR software library.

psmConnect

int psmConnect(char *host, int port)

Connect to state manager server.

Parameters:

host - the name of the state manager server host.
port - TCP port for the state manager server. Zero to use default port.

Returns:

TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmDestroyObject

int psmDestroyObject(char *object)

Destroy an object that was created by psmNewObject.

Parameters:

object - the name of the object to be destroyed.

Returns:

TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmDisconnect

int psmDisconnect(void)

Disconnect from the state manager server.

Returns:

TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmEndActivity

int psmEndActivity(char *object, char *state, char *event, char * logmsg)

End activity for an object and cause a state transition.

Parameters:

object - the name object that is ending an activity.
state - current state of the object before the transition.
event - event that ends the activity.
logmsg - message to enter in message log describing end of activity.

Returns:
TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmErrorCode

int psmErrorCode()

Get the error code for the last error detected during a state manager API call.

Returns:
Error code (see psmcodes.h).

psmGetModel

int psmGetModel(char *object, char *model)

Get the name of the state model for an object.

Parameters:
object - name of object for request.
model - returned model for object.

Returns:
TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmGetSocket

int psmGetSocket(SM_SOCKET *pSocket)

Get the socket that is connected to the state manager.

Parameters:
pSocket - pointer to return location for socket.

Returns:
TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmGetState

int psmGetState(char *object, char *state)

Get the current state of an object.

Parameters:
object - name of the object for request.
state - returned state for object.

Returns:
TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmNewObject

int psmNewObject(char *object, char *model, char *logmsg)

Create a new object with a specified state model.

Parameters:
object - the name object to be created.

model - state model for the object.
logmsg - message to enter in message log.

Returns:
TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmPerror

void psmPerror(char *msg)

Print an error message for the last error detected during an API call.

Parameters:
msg - text to be printed before the error message.

psmRecvEvent

int psmRecvEvent(char *object, char *event, char * arg, int *pTimedOut, int waitTimeMsec)

Receive an event for an object and do a state transition.

Parameters:
object - returned object name.
event - returned event for the object.
arg - returned event argument.
pTimedOut - location to return timed out flag (TRUE if timed out else FALSE).
waitTimeMsec - time to wait in msec before timeout.

Returns:
TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().

psmStrToCode

int psmStrToCode(SM_STR_CODE_T *pCode, char *str)

Convert a string for an event or state to a enum code. See psmcodes.h.

Parameters:
pCode - location to return value for code.
str - string to be converted to code.

Returns:
TRUE for success or FALSE for failure. See psmErrorCode() and psmPerror().