Tekkotsu Behaviors & Events

15-494 Cognitive Robotics
David S. Touretzky &
Ethan Tira-Thompson

Quiz (1)

- Given these node definitions:

  hi: SpeechNode(“Hello”)
  bye: SpeechNode(“Goodbye”)
  ping: SoundNode(“ping.wav”)

- What's the difference between this:

  hi =C=> bye
  hi =C=> ping

- And this:

  hi =C=> {bye, ping}

Quiz (2)

- Given this node class definition:

```
$nodeclass MyThing : StateNode : doStart {
  ...
}
```

- What's the difference between:

```
thing: MyThing =C=> OtherThing =C=> MyThing
```

- And this:

```
thing: MyThing =C=> OtherThing =C=> thing
```

Disclaimer

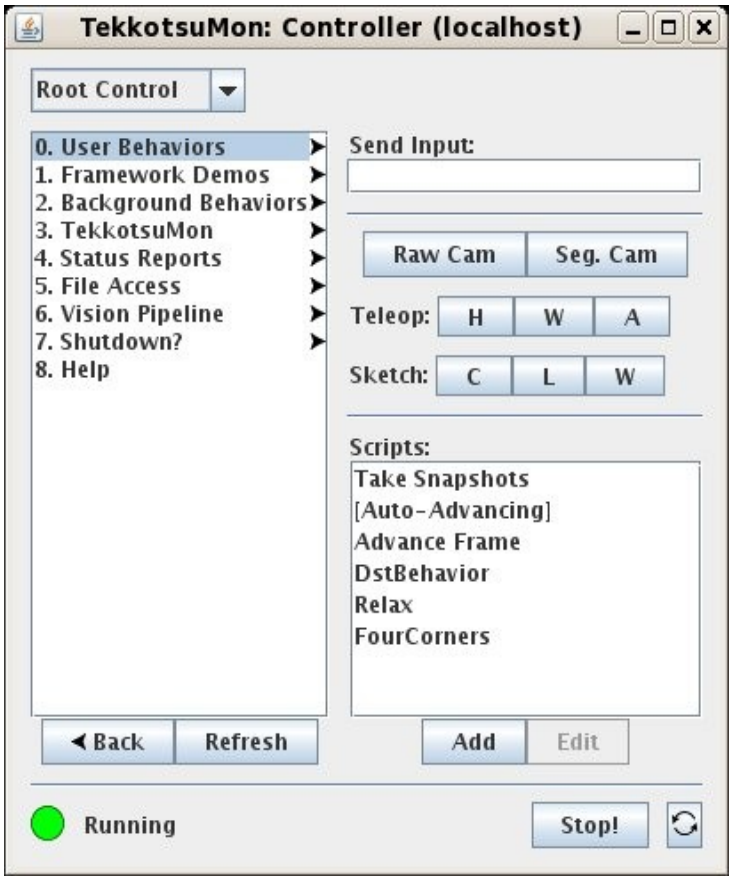
- This lecture will show you how Tekkotsu works at the basic level of behaviors and events.
- Some slides will contain...

ugly computer source code.

- Tekkotsu programmers don't really code this way.
- They use the state machine shorthand instead.

Behaviors

- State machines are behaviors.
  - Both state nodes and transitions are behaviors.
- Behaviors are instances of *classes*.
  - Add them to the ControllergGUI "User Behaviors" menu using the REGISTER_BEHAVIOR macro.
  - Double click on the "User Behaviors" menu item to instantiate and run.
  - When you stop a behavior (double click on the menu item again), the instance is deleted.



Five Behavior Components

```
#include “Behaviors/BehaviorBase.h”

class PoodleBehavior : public BehaviorBase {
```

- Constructor

```
  PoodleBehavior(const std::string &name) :
    BehaviorBase("PoodleBehavior") {}
```

- doStart() is called when the behavior is activated

```
  virtual void doStart() {
    cout << getName() << " is starting up." << endl;
  }
```

Five Behavior Components

- doStop() is called when the behavior is deactivated, but you rarely need to bother with this.

```
virtual void doStop() {
   cout << getName() << " is shutting down." << endl;
 }
```

- doEvent processes requested event types

```
virtual void doEvent() {
   cout << getName() << " got event: "
        << event->getDescription() << endl;
 }
```

Five Behavior Components

- getClassDescription() returns a string displayed by ControllerGUI pop-up help

```
    static std::string getClassDescription() {
      return "Demonstration of a simple behavior";
    }


};  // end of PoodleBehavior class definition
```

Behaviors are Coroutines

- Behaviors are coroutines, not threads:
  - Many can be “active” at once, but...
  - Only one is actually running at a time.
  - No worries about mutual exclusion.
  - Must voluntarily relinquish control so that other active behaviors can run.
- BehaviorBase is a subclass of:
  - EventListener
  - ReferenceCounter
- Behaviors will be deleted if they are deactivated and the reference count goes to zero.

Browsing the Documentation

- Go to Tekkotsu.org and click on “Reference” in the gray nav bar.

- “Class List” in the left nav bar
  - Click on a class name (`BehaviorBase`) to see documentation
  - Then click on a method name (`doEvent`) to jump to detailed description
  - Click on line number to go to source code

- “Directories” in left nav bar shows major components
  - Look at the `Behaviors` and `Events` directories

Searching the Source

- The “search” box in the online documentation can be used to search for classes, methods, variables, enumerated types, etc.

- Use the “ss” shell script to search the source code using grep:

```
> cd /usr/local/Tekkotsu

> ss LBump

> ss IRDist
```

Events

- Events are subclasses of `EventBase`
- Three essential components:

1. Generator ID: what kind of event is this?

   buttonEGID, visionEGID, timerEGID, ...

2. Source ID: which sensor/actuator/behavior/thing generated this event?

   CreateInfo::PlayButOffset
   ERS7Info::HeadButOffset

3. Type ID, which must be one of:

   activateETID
   statusETID
   deactivateETID

Where Are These Defined?

- EventGeneratorID_t defined in Events/EventBase.h
- Event source ids are specific to the event type:
  - PlayButOffset defined in Shared/CommonCalliopeInfo.h
  - visPinkBallSID defined in Shared/ProjectInterface.h
  - For completion events, the source id is the address of the state node that is completing.
- EventTypeID_t defined in Events/EventBase.h

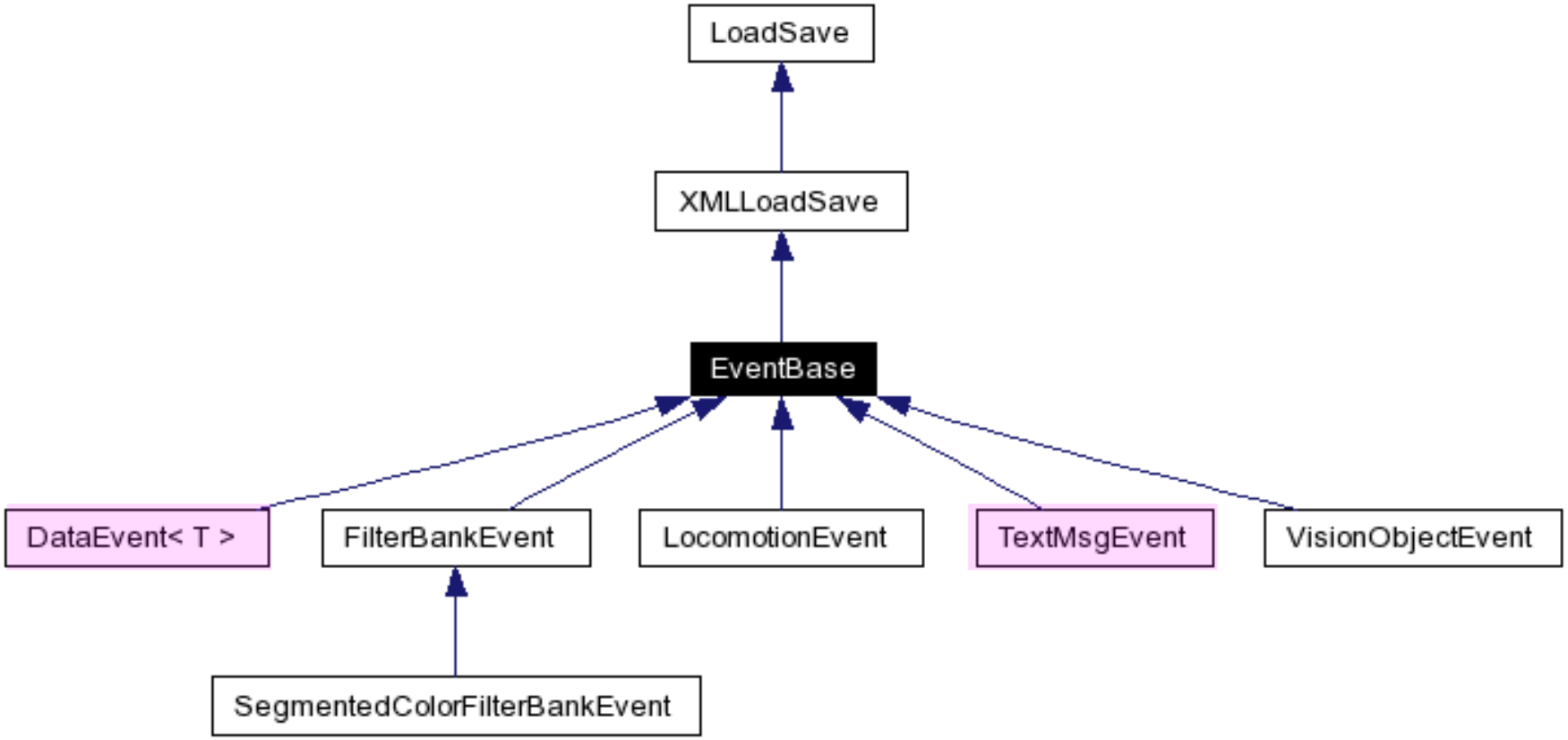
```
enum EventTypeID_t {
    activateETID,
    statusETID,
    deactivateETID,
    numETIDs
  };
```

Event Generator IDs

unknownEGID
aiEGID
audioEGID
buttonEGID
cameraResolutionEGID
erouterEGID
estopEGID
grasperEGID
locomotionEGID
lookoutEGID
mapbuilderEGID
micOSndEGID
micRawEGID
micFFTEGID
micPitchEGID
mocapEGID
motmanEGID
pilotEGID
powerEGID
remoteStateEGID
runtimeEGID
sensorEGID
servoEGID
stateMachineEGID
stateSignalEGID
stateTransitionEGID
textmsgEGID
timerEGID
userEGID
visInterleaveEGID
visJPEGEGID
visObjEGID
visOFbkEGID
visPNGEGID
visRawCameraEGID
visRawDepthEGID
visRegionEGID
visRLEEGID
visSegmentEGID
wmVarEGID
worldModelEGID

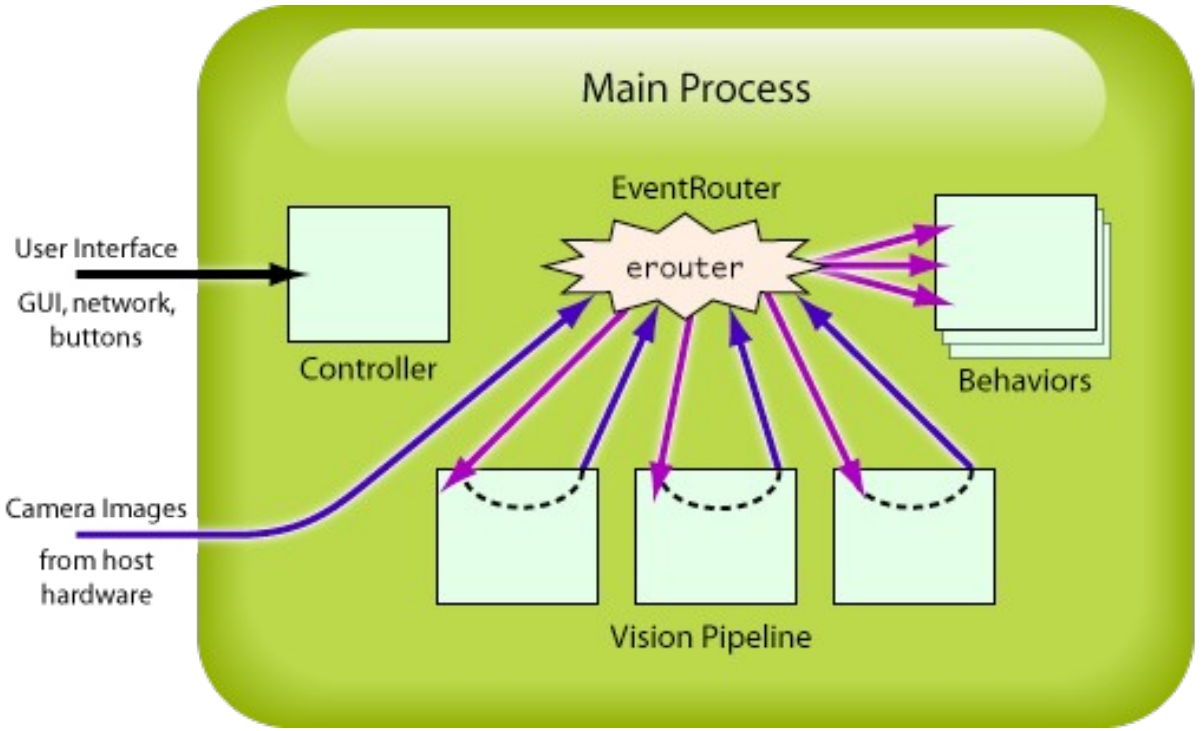
Types of Events

- Most events are described using EventBase.
- A few specialized events require additional fields to convey all their information, so they use a specialized subclass of EventBase.



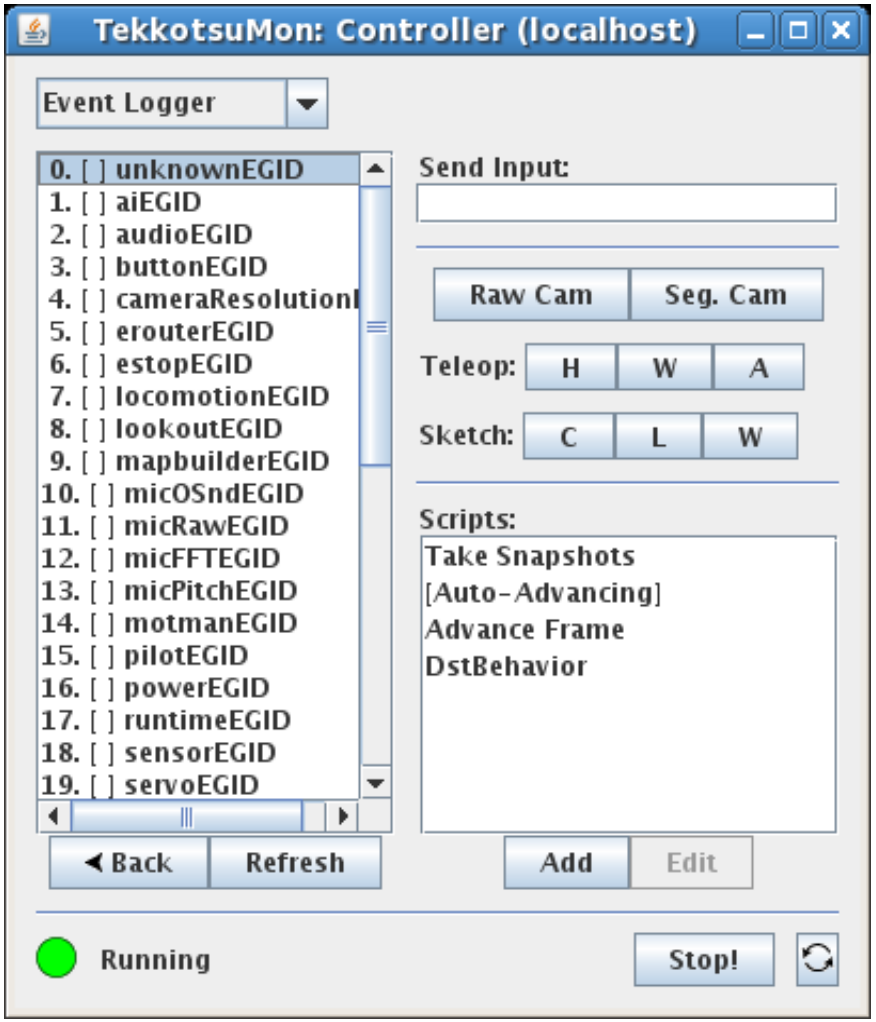
The Event Router

- Runs in the Main process.
- Distributes events to the Behaviors listening for them.



The Event Logger

- Root Control
  > Status Reports
  > Event Logger
- Outputs to console
- Use shift-click to select a range of entries.



Subscribing to Events

addListener(*listener*, *generator*, *source*, *type*)

```
#include “Events/EventRouter.h”

virtual void doStart() {
  erouter->addListener(this,
                       EventBase::buttonEGID,
                       RobotInfo::GreenButOffset,
                       EventBase::activateETID);
}
```

Transitions do this to listen for events, so you don't have to call addListener() yourself.

Processing Events

```
virtual void doEvent() {
  switch ( event->getGeneratorID() ) {

    case EventBase::buttonEGID:
      cout << "Button press: " << event->getDescription()
           << endl;
      break;

    default:
      cout << "Unexpected event: "
           << event->getDescription() << endl;
  }
}
```

Transitions use doEvent() to check the event and decide whether to fire.

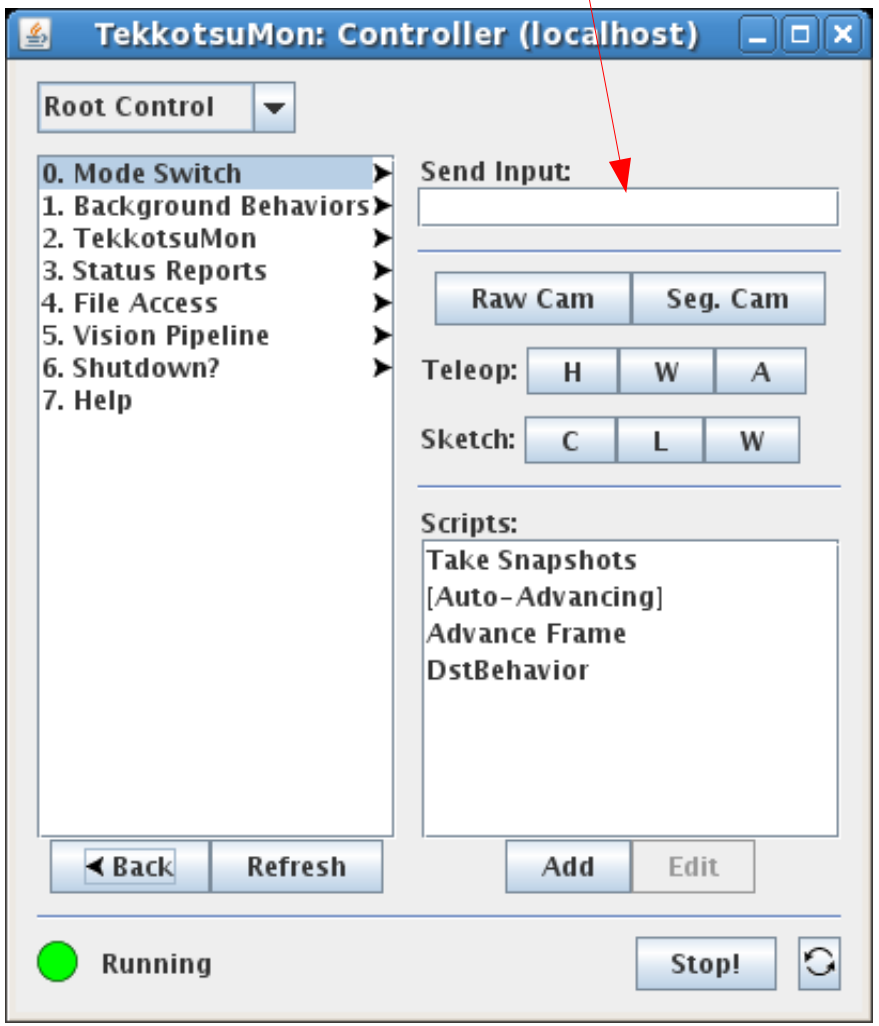
Text Message Events

You can send text messages to the robot via the ControllerGUI's "Send Input" window:

```
!msg Hi there
```

This causes the behavior controller to post a TextMsgEvent.

You can also give the msg command to Tekkotsu's command line (with no exclamation point).



Subscribing to TextMsg Events

```
#include "Events/TextMsgEvent.h"

virtual void doStart() {
  erouter->addListener(this, EventBase::textmsgEGID);
}
```

The source ID is meaningless (it's -1).

The type ID is always statusETID.

Casting TextMsg Events to Get Access to the String

```
void doEvent() {
  switch ( event->getGeneratorID() ) {

  case EventBase::textmsgEGID: {
     const TextMsgEvent *txtev =
          dynamic_cast<const TextMsgEvent*>(event);
     cout << "I heard: '” << txtev->getText() << "'" << endl;
     };
     break;

  case EventBase::buttonEGID:
    ...
```

State Machine Shorthand for Text Message Events

```
waitForUser: StateNode
waitForUser  =TM("cheeseburger")=>  giveBurger
waitForUser  =TM("fries")=>  giveFries
waitForUser  =TM=>  askAgain
```

Competing transitions can fire in any order, and the first one “wins”. So how does the default =TM=> case work?

Answer: a timer delays firing so the other transitions can fire first if they match the string.

Timers

Timers are good for two kinds of things:

- Repetitive actions: “Bark every 30 seconds.”
  - Whenever a timer expires and a timer expiration event is posted, the timer should be automatically restarted.
- Timeouts: “If you haven't seen the ball for 5 seconds, bark and turn around.”
  - One-shot timer. Will need to be cancelled if we see the ball before the time expires.

addTimer

- addTimer(*listener*, *source*, *duration*, *repeat*)
  - listener is normally this
  - source is an arbitrary integer
  - duration is in milliseconds
  - repeat should be “true” if a sequence of timer events is desired
- Starts timer and automatically listens for the event.
- Timers are specific to a behavior instance; can use the same source id in other behaviors without interference.
- Behaviors can receive another's timer events if they use addListener to explicitly listen for them.
- removeTimer(*listener*, *source*)

Timer Example

```
#include “Behaviors/BehaviorBase.h”
#include “Events/EventRouter.h”

virtual void doStart() {

  erouter->addListener(this,
                       EventBase::buttonEGID,
                       RobotInfo::PlayButffset,
                       EventBase::activateETID);

  erouter->addListener(this,
                       EventBase::buttonEGID,
                       RobotInfo::AdvanceButOffset,
                       EventBase::activateETID);
}
```

Timer Example

```
virtual void doEvent() {
  switch ( event->getGeneratorID() ) {

  case EventBase::buttonEGID:
    if ( event->getSourceID() == RobotInfo::PlayButOffset )
      erouter->addTimer(this, 1234, 5000, false);
    else if (event->getSourceID() == RobotInfo::AdvanceButOffset)
      erouter->removeTimer(this, 1234);
  break;

  case EventBase::timerEGID:
    cout << “On no!!!!  Timer expired!” << endl;
  }

}
```

What does this behavior do?

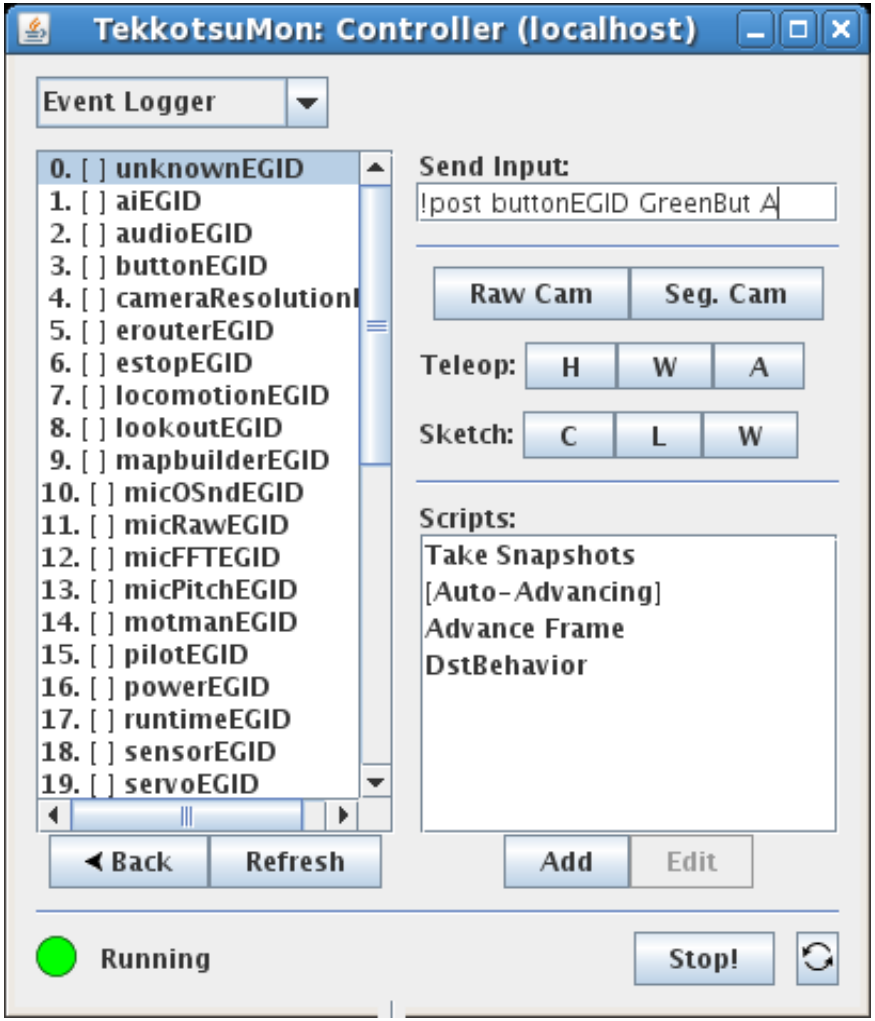
How would you implement this functionality using the state machine mechanism?

ControllerGUI Can Post Events To Tekkotsu

Type this command in the "Send Input" box:

```
!post buttonEGID Play A
```

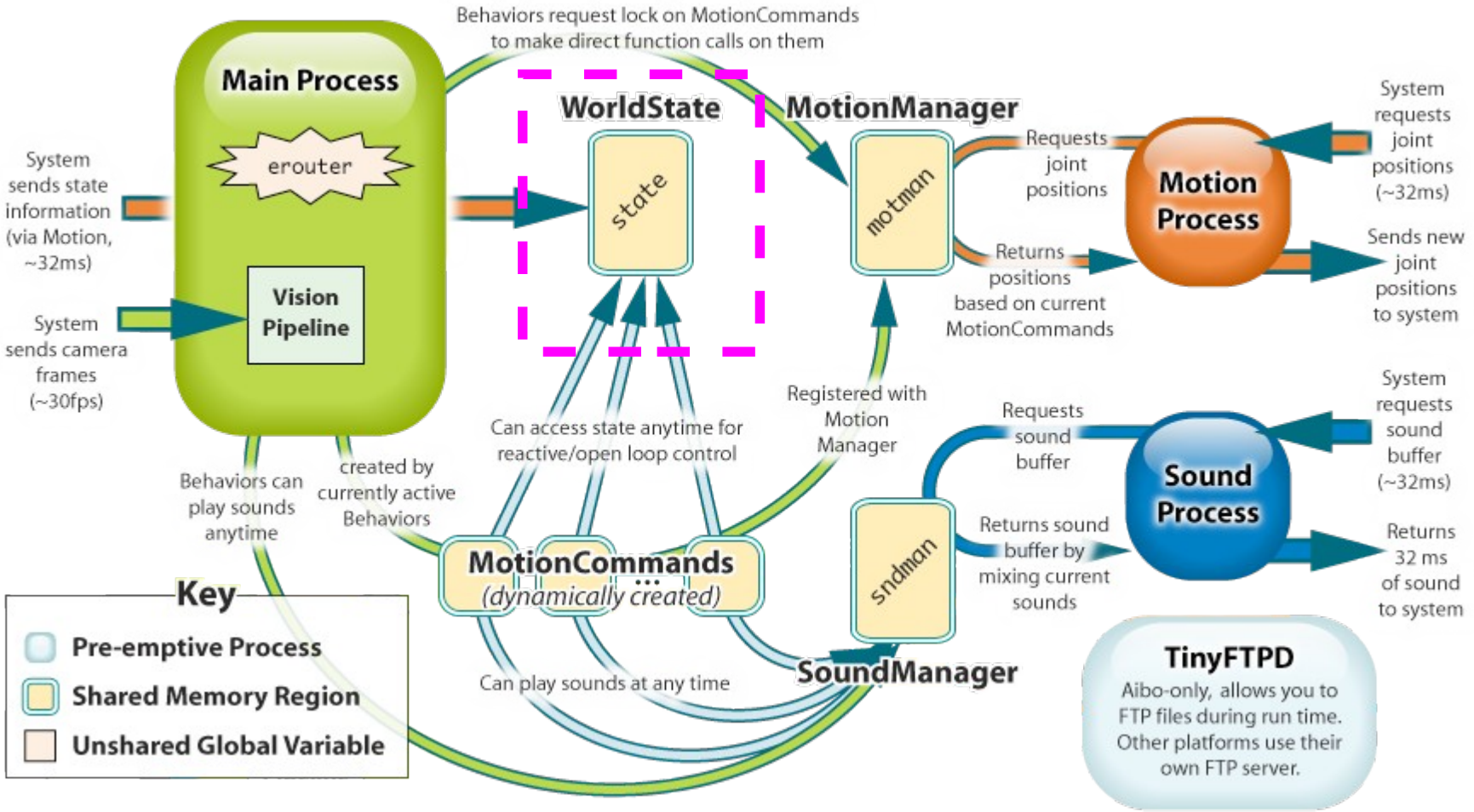
- Monitor the result using the Event Logger
- You can also use the post command in the Tekkotsu command line (no exclamation point).



What Is A Completion Event?

- State nodes use completion events to indicate that their action has completed successfully.
- Event content:
  - Generator id: stateMachineEGID
  - Source id: address of the state node that is completing
  - Type id: statusETID
- CompletionTrans looks for completion events. Shorthand form: =C=>
- If you define your own node class as a subclass of StateNode, you can signal completion by calling postStateCompletion().

Tekkotsu Architecture



World State

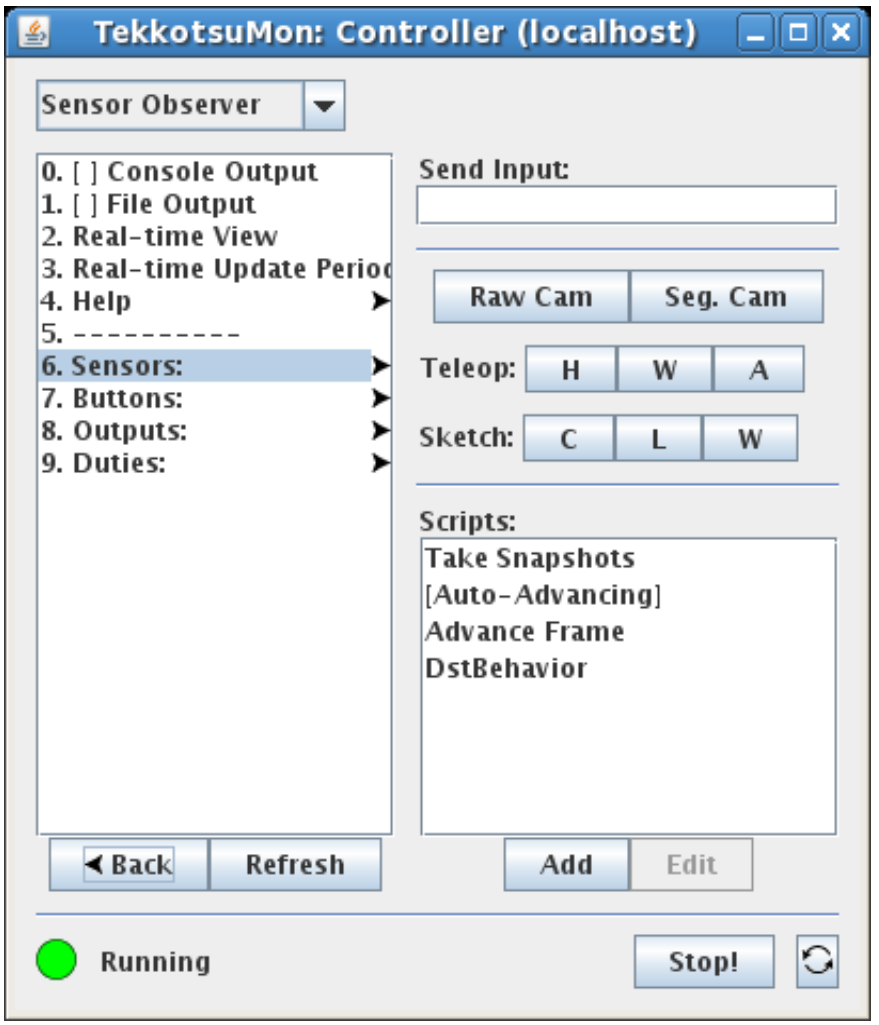
- Shared memory structure between Main and Motion
- Updated every 32 msec
- sensorEGID events announce each update
- Contents:
  - joint positions, duty cycles, and PID settings
  - button states: `state->buttons[PlayButOffset]`
  - IR range readings: `state->sensors[CenterIRDistOffset]`
  - accelerometer readings (if installed)
  - battery state, thermal sensor
  - commanded walking velocity (x,y,a)

Sensor Observer Monitors the Sensor Portion of World State

- Root Control
  > Status Reports
  > Sensor Observer

- Try monitoring the IR wall sensor.

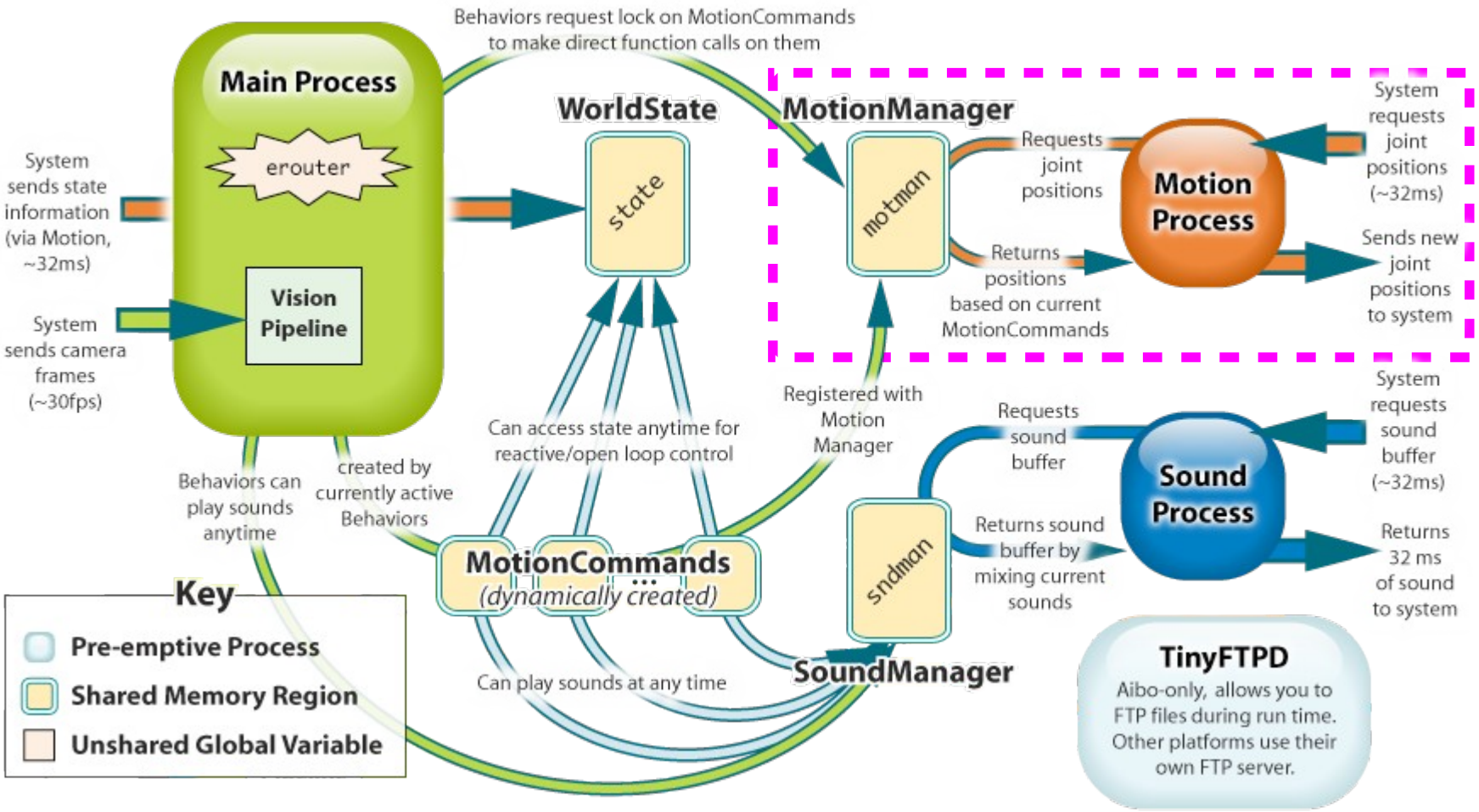
- Then move your hand in front of the robot.



Control of Effectors

- How do we make the robot move?
- Must send commands to each device (head, legs, arm, LED display, etc.) every 32 ms.
- This is real-time programming.
- Can't spend too long computing command values!
- Best to do all this in another process, independent of user-written behaviors, so motion can be smooth.

Tekkotsu Architecture: Motion



Motion Command State Nodes

- WalkNode, ArmNode, HeadPointerNode, LEDNode, etc...
- Creates the motion command in shared memory.
- User can “program” the motion command by calling one of its methods to tell it what to do.
- The node's start() method registers the motion command with the Motion Manager, making it active.
- The node listens for motion manager events to detect when the motion is complete.
- Removes the motion command when it completes.
- Posts a completion event to notify the outgoing =C=> transition to fire.

Modern Tekkotsu Programming

- Control structure provided by state machine language.
  - Events and listeners are handled for you.
- Much reliance on “the Crew”:
  - MapBuilder for vision
  - Pilot for navigation and localization
  - Grasper for manipulation
  - Lookout for control of the sensor package
- User applications build on these primitives and extend them where necessary.