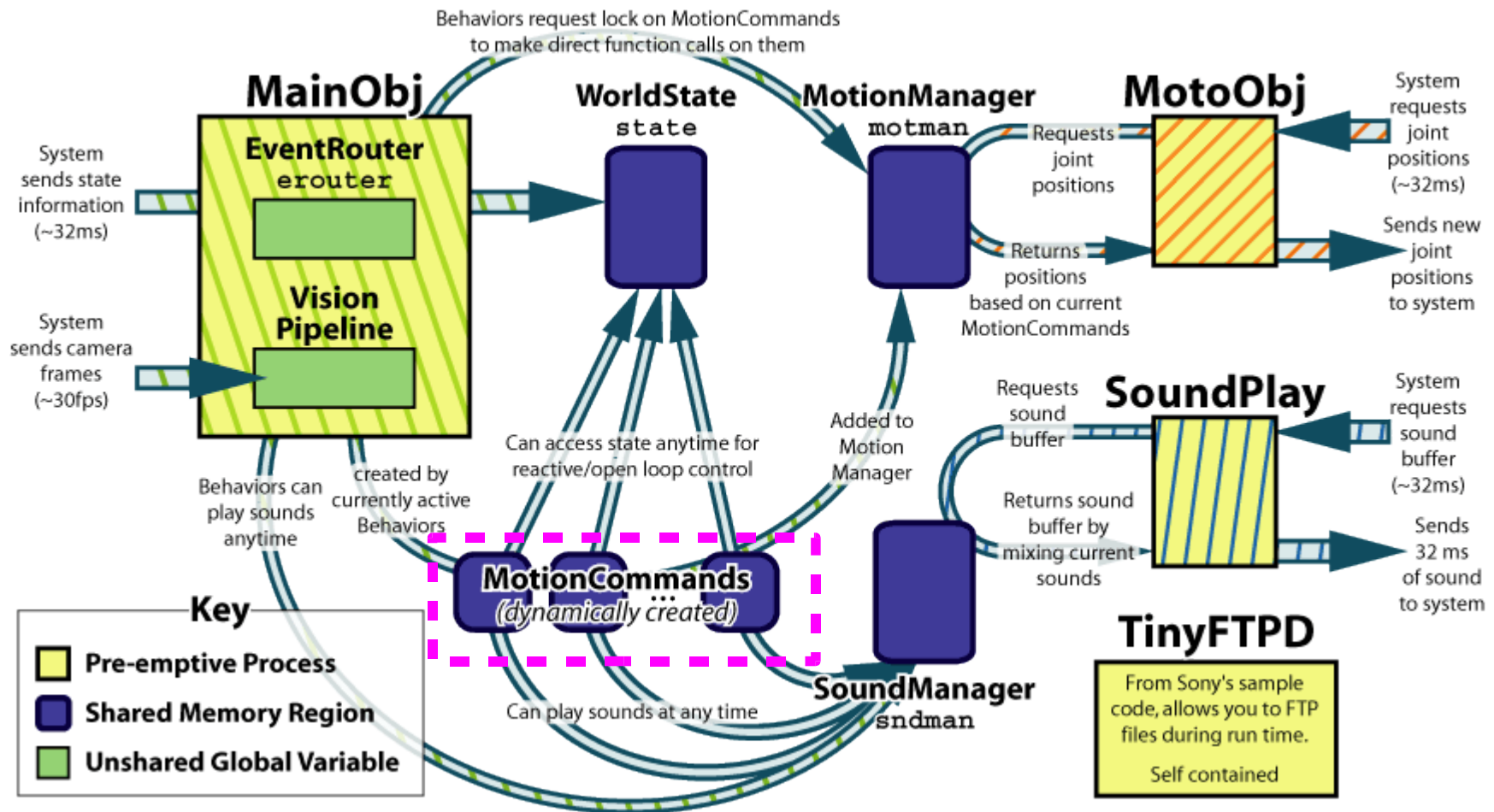


Motion Commands and Real-Time Programming

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

Carnegie Mellon
Spring 2007

Motion Commands Live in Shared Memory

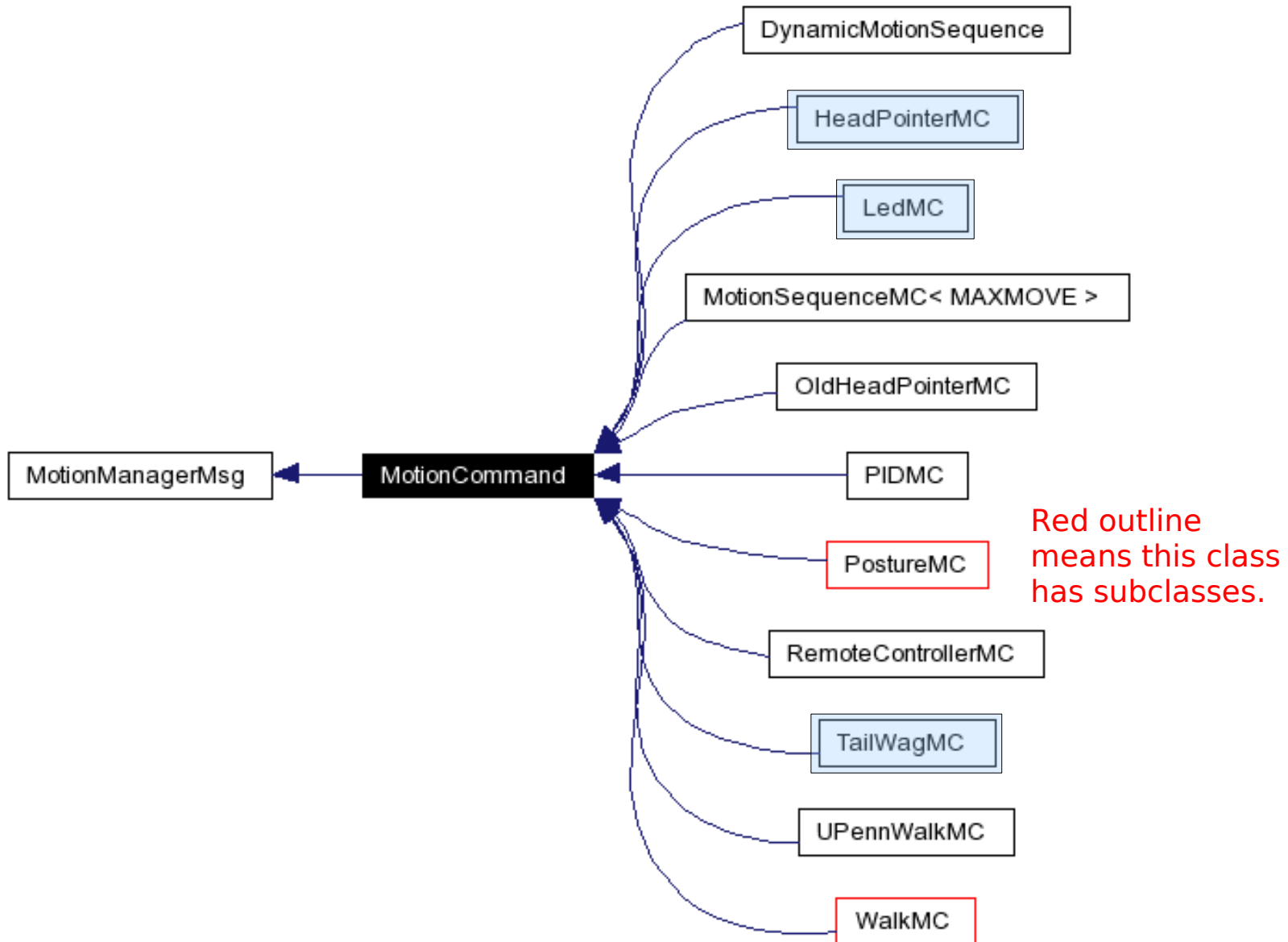


Motion Commands Are Objects

A MotionCommand is an object with 2 kinds of methods:

- 1) Command methods for telling it what you want it to do.
 - Called by user code running in Main.
- 2) An `updateOutputs()` method for computing new effector values (joint angles, LED brightness, etc.)
 - Called every 32 ms by the motion manager, running in Motion.

Types of Motion Commands

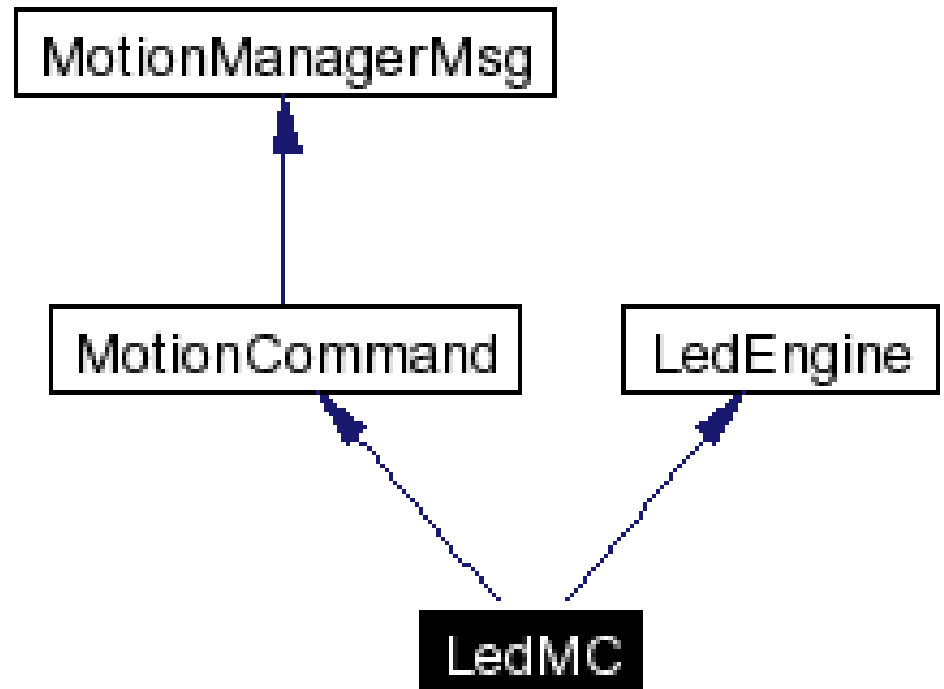


Creating a Motion Command

- `SharedObject<LedMC> leds_mc;`
- The actual LedMC object is created in shared memory.
- The SharedObject `leds_mc` lives in Main's address space, and holds a pointer to the shared memory region.
- Two ways to refer to a motion command within Main:
 - via the shared object
 - via the MC_ID (Motion Command ID) assigned to it by the Motion Manager (motman) when the motion command is active

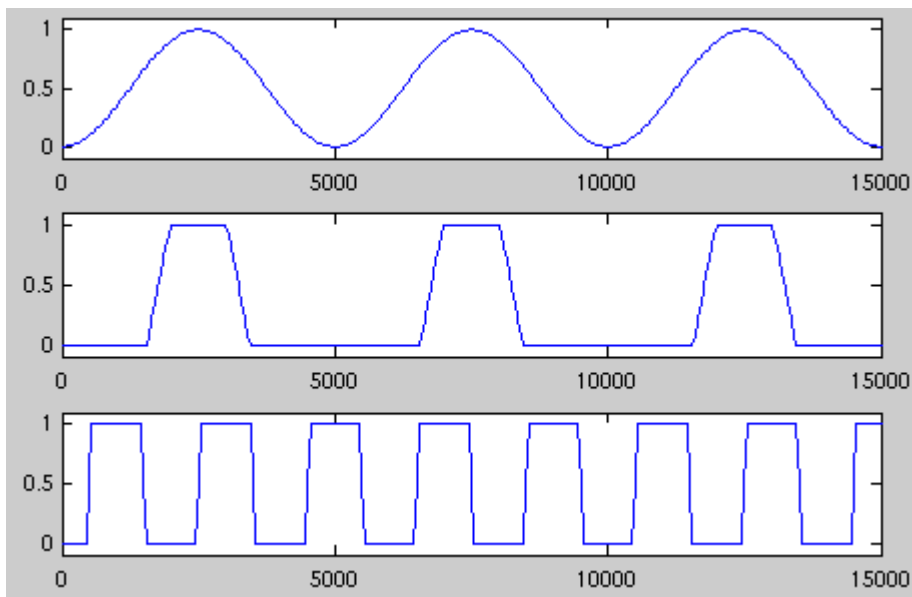
LedMC

- Defined in Motion/LedMC.h
- LedMC inherits from two parent classes.
- MotionCommand:
 - updateOutputs()
 - isAlive()
 - isDirty()
- LedEngine:
 - cycle(...)
 - flash(...)
 - invert(...)
 - etc.



LedEngine

- `cycle(LEDBitmask_t bitmask,
unsigned int period,
float amplitude,
float offset=0,
int phase=0)`



period = 5000 ms
amplitude = 1

period = 5000 ms
amplitude = 5
offset = -1

period = 2000 ms
amplitude = 200



Sample LedMC Program

```
#include "Behaviors/BehaviorBase.h"
#include "Motion/LedMC.h"
#include "Motion/MotionManager.h"

class DstBehavior : public BehaviorBase {

protected:
    MotionManager::MC_ID leds_id;    // id of MotionCommand

public:
    DstBehavior() : BehaviorBase("DstBehavior"),
                   leds_id(MotionManager::invalid_MC_ID) {}
}
```


Sample LedMC Program

```
virtual void DoStart() {  
  
    BehaviorBase::DoStart();  
    cout << getName() << " is starting up." << endl;  
  
    SharedObject<LedMC> leds_mc;  
    leds_mc->cycle(RobotInfo::FaceLEDMask, 1000, 100.0);  
    leds_id = motman->addPersistentMotion(leds_mc);  
  
}
```

- SharedObjects and MCs are both reference counted.
- What happens to `leds_mc` when `DoStart` returns?
- What happens to the motion command?

Operator Overloading

- `leds_mc` is of type `SharedObject`
- `cycle(...)` is a method of `LedEngine`, not `SharedObject`.
- So why does this work?

```
leds_mc -> cycle(RobotInfo::FaceLEDMask, 1000, 100.0);
```

- The arrow operator is overloaded by `SharedObject`. It will dereference the pointer to the actual `LedMC` in shared memory, and call its `cycle(...)` method.

Sample LedMC Program

```
virtual void DoStop() {  
  
    motman->removeMotion(leds_id);  
    leds_id = MotionManager::invalid_MC_ID;  
  
    cout << getName() << " is shutting down." << endl;  
    BehaviorBase::DoStop();  
  
}
```

- We needed to keep leds_id around so we could reference the motion command in DoStop().
- You should always remove motion commands when you're done with them, unless autopruned.
- cycle() can't be autopruned. Why not?

Mutual Exclusion: MMAccessor

- Suppose we want to change the parameters of a motion command while it's active.
- Example: change the cycle period of a LedMC.
- Not safe for Main to change an MC while Motion is trying to use it. Need a mutex mechanism:

```
{  
    MMAccessor<LedMC> leds_acc(leds_id);  
    leds_acc->cycle(RobotInfo::FaceLEDMask, 250, 100.0);  
}
```

- Constructor handles checkout; destructor handles checkin. Within scope of leds_acc, motman locked out.
- Don't lock it out for too long!

Changing the Cycle Period When a Button Is Pressed

```
virtual void DoStart() {  
  
    BehaviorBase::DoStart();  
    cout << getName() << " is starting up." << endl;  
  
    SharedObject<LedMC> leds_mc;  
    leds_mc->cycle(RobotInfo::FaceLEDMask, 1000, 100.0);  
    leds_id = motman->addPersistentMotion(leds_mc);  
  
    erouter->addListener(this,  
                          EventBase::buttonEGID,  
                          RobotInfo:LFrPawOffset);  
  
}
```

Changing the Cycle Period When a Button Is Pressed

```
virtual void processEvent(const EventBase &event) {  
    int new_period = 1000 - 750*event.getMagnitude();  
  
    MMAccessor<LedMC> leds_acc(leds_id);  
    leds_acc->cycle(RobotInfo::FaceLEDMask,new_period,100.0);  
}
```

A ShortCut

- Use this to change multiple MC parameters:

```
MMAccessor<LedMC> leds_acc(leds_id);  
leds_acc->cycle(RobotInfo::FaceLEDMask,500,1.0);  
leds_acc->cycle(RobotInfo::BackLEDMask,2000,2.0);
```

- Use this if you only need to change one:

```
MMAccessor<LedMC>(leds_id)->  
    cycle(RobotInfo::FaceLEDMask,500,1.0);
```

Prunable Motions

- `flash(LEDBitMask_t bitmask,
float value,
unsigned int msec)`

Sets the specified LEDs to *value* for so many msec, then sets them back.

- Once the action is complete, the motion command has no more work to do.
- If it's a persistent motion, it sits around waiting for its next assignment. If a prunable motion, the motion manager removes (prunes) it.

Flash the Back LEDs for 15 secs

```
virtual void DoStart() {  
    BehaviorBase::DoStart();  
  
    SharedObject<LedMC> leds_mc;  
    leds_mc->flash(RobotInfo::BackLEDMask, 15000, 1.0);  
    leds_id = motman->addPrunableMotion(leds_mc);  
    cout << "Created LedMC, id = " << leds_id << endl;  
  
    BehaviorBase::DoStop();  
}
```

- What if you started and stopped this behavior several times within a 15 second period?

Naming of Effectors

- Tekkotsu maintains many arrays of effectors: joint angles, joint PID settings, output ranges, LED values, etc.
- Effectors are named by their offset into these arrays.
- See definition of LEDOffset_t in ERS7Info.h

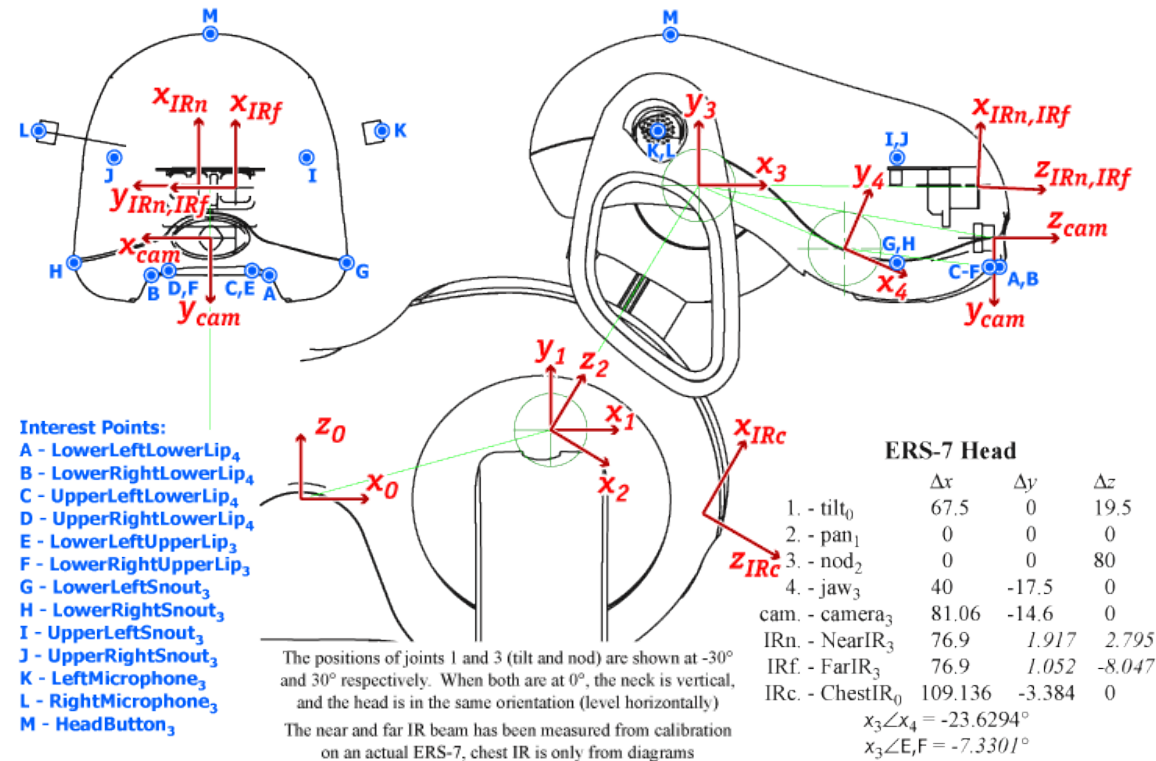
```
enum LEDOffset_t {  
    HeadColorLEDOffset = LEDOffset,  
    ...  
    FaceLEDPanelOffset,  
    ... };
```

- Bitmasks are derived from the offsets:

```
const LEDBitMask_t FaceLEDPanelMask =  
    1 << (FaceLEDPanelOffset-LEDOffset);
```

Moving the Head

- Three head joints:
tilt, pan, nod
- Head effector names:
TiltOffset
PanOffset
NodOffset
- HeadOffset too:
outputRanges[HeadOffset+TiltOffset]



HeadPointerMC

Defined in Motion/HeadPointerMC.h

- void setJoints(float *tilt*, float *pan*, float *nod*)
- void setJointValue(unsigned int *j*, float *value*)
 - setJointValue(TiltOffset, 0.5)
- float getJointValue(unsigned int *j*) const
- void setMaxSpeed(unsigned int *j*, float *x*)

Detecting Motion Completion

- It takes time to move the head.
- Behaviors can't wait around: must relinquish control! (If they don't, sensor values can't be updated, since this happens in the same process, Main, where the behaviors run.)
- HeadPointerMC posts a status event when motion completes or times out. The generator is motmanEGID.
- To smoothly chain actions together, listen for status events. (Or use Tekkotsu's state machine formalism.)
- Example: moving the head and then blinking.

Move Head Then Blink

```
#include "Behaviors/BehaviorBase.h"
#include "Events/EventRouter.h"
#include "Motion/HeadPointerMC.h"
#include "Motion/LedMC.h"
#include "Motion/MMAccessor.h"
#include "Motion/MotionManager.h"
#include "Shared/WorldState.h"

class DstBehavior : public BehaviorBase {
protected:
    MotionManager::MC_ID leds_id, head_id;

public:
    DstBehavior() : BehaviorBase("DstBehavior"),
        leds_id(MotionManager::invalid_MC_ID),
        head_id(MotionManager::invalid_MC_ID) {}
}
```

Move Head Then Blink

```
virtual void DoStart() {  
    BehaviorBase::DoStart();  
  
    SharedObject<LedMC> leds_mc;  
    leds_id = motman->addPersistentMotion(leds_mc);  
  
    SharedObject<HeadPointerMC> head_mc;  
    head_mc->setMaxSpeed(RobotInfo::TiltOffset,0.5);  
    head_id = motman->addPersistentMotion(head_mc);  
  
    erouter->addListener(this,EventBase::buttonEGID);  
  
    erouter->addListener(this,  
                        EventBase::motmanEGID,  
                        head_id,  
                        EventBase::statusETID);  
}
```

Move Head Then Blink

```
virtual void DoStop() {  
  
    motman->removeMotion(leds_id);  
    leds_id = MotionManager::invalid_MC_ID;  
  
    motman->removeMotion(head_id);  
    head_id = MotionManager::invalid_MC_ID;  
  
    BehaviorBase::DoStop();  
  
}
```


Move Head Then Blink

```
float calcNewHeadTarget() {  
    const float lowpos =  
        RobotInfo::outputRanges[HeadOffset+TiltOffset]  
            [MinRange];  
    const float highpos =  
        RobotInfo::outputRanges[HeadOffset+TiltOffset]  
            [MaxRange];  
    const float midpos = (lowpos + highpos) / 2;  
  
    const float curpos = state->outputs[HeadOffset+TiltOffset];  
  
    if ( curpos < midpos )  
        return highpos;  
    else  
        return lowpos;  
}
```

Move Head Then Blink

```
virtual void processEvent(const EventBase &event) {  
    switch ( event.getGeneratorID() ) {  
  
        case EventBase::buttonEGID:  
            if ( event.getTypeID() == EventBase::activateETID )  
                MMAccessor<HeadPointerMC>(head_id)->  
                    setJointValue(TiltOffset, calcNewHeadTarget());  
            break;  
  
        case EventBase::motmanEGID:  
            MMAccessor<LedMC>(leds_id)->  
                flash(RobotInfo::FaceLEDMask, 1000);  
    }  
}
```

Thought Questions

- 1) Suppose you push a button, the head starts to move, and you push the button again. What happens?
- 2) Suppose you activate the behavior, then turn on HeadPointerRemoteControl and try to move the head around. The result is jerky and the motion is attenuated. Why?
- 3) Suppose you don't want your active HeadPointerMC to actually affect the head until the user presses a button? What are some ways you could prevent this?

Motion Command Priority Level

- `kIgnoredPriority = -1.0` won't be expressed
- `kBackgroundPriority = 0.0` use if nothing else running
- `kLowPriority = 5.0`
- Default: `kStdPriority = 10.0` what you get by default
- `kHighPriority = 50.0`
- `kEmergencyPriority = 100.0` used by Emergency Stop

Move and Then Blink

- In DoStart()

```
head_mc->setPriority(kIgnoredPriority);
```

- In processEvent()

```
head_acc->setPriority(kStdPriority);
```

- Note: setPriority() is inherited from MotionCommand.
- Does not show up in the method list in the documentation for HeadPointerMC.

Motion Command Weight

- For each joint,, the Motion Manager orders commands by priority and computes a weighted average as a function of both the priorities and the weights.
- Starting with the highest priority, if weights of active motion commands sum to < 1 , the remaining weight is allocated to the next highest priority, and so on.
- Weights are adjustable. To set tilt/pan/nod weights:

```
head_mc->setWeight(0.5)
```

- Need to set individual joint weights? Use a PostureMC.

TailWagMC

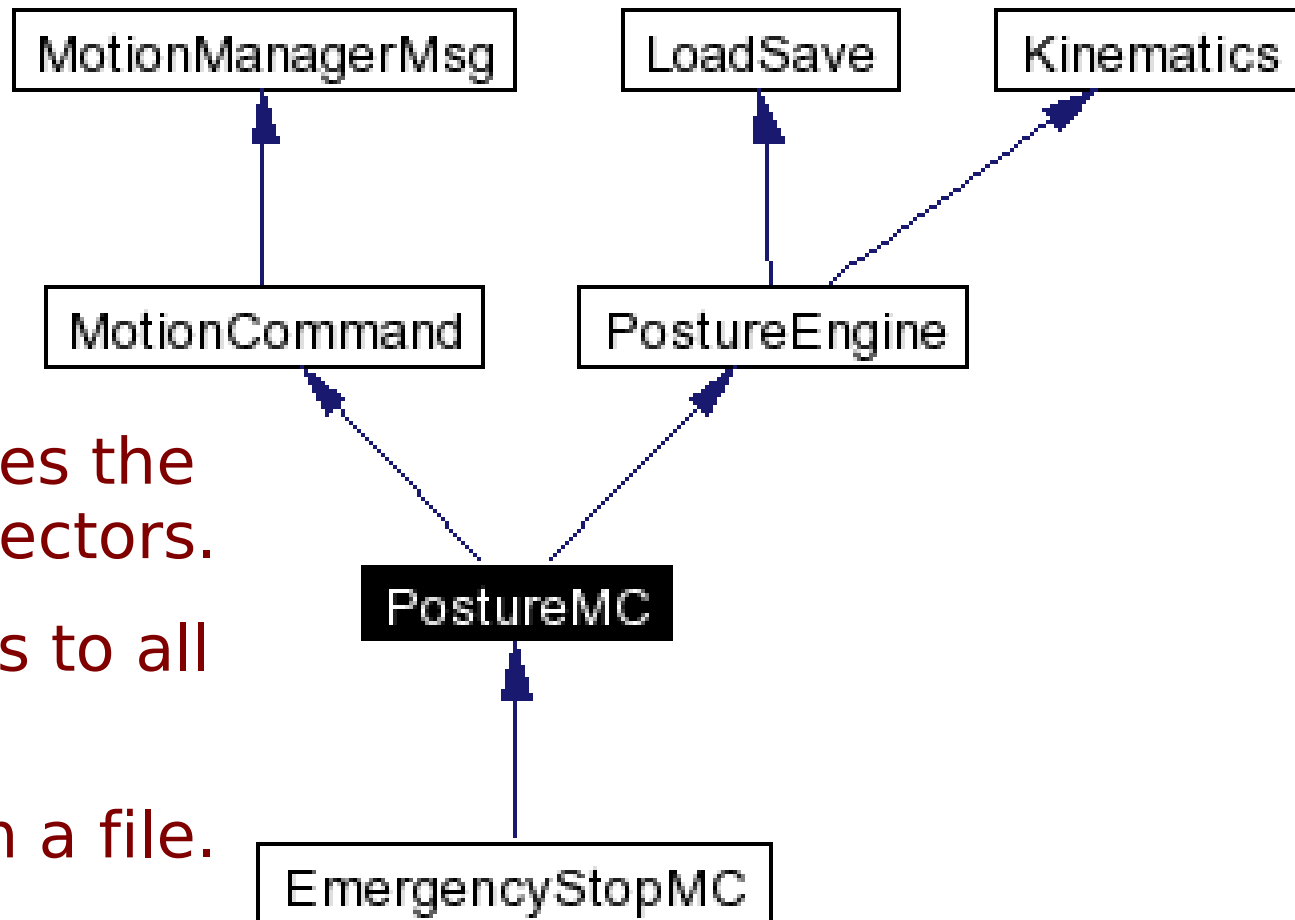
- Wags tail back and forth (sine wave).
- User-specified period, magnitude.
- User can also adjust the tilt (but this doesn't change during wagging.)
- Stop/start with setActive(bool)
- Can “unset” the tilt to allow some other motion command to control it while this one handles the wag.
- “Unset” = set tail tilt weight to zero.



How MCs Really Work

- LEDs, head joints, tail joints, etc. are all effectors.
- OutputCmd specifies a value and weight for one effector.
- AperiOS wants new effector values every 8 msec. But it buffers them 4 frames at a time.
- So LedMC, HeadPointerMC, and TailWagMC are called every 32 msec and need to return 4 frames' worth of output.
- HeadPointerMC uses 3 OutputCmds, one per joint.
LedMC uses 4×27 OutputCmds (there are 27 LEDs).
TailWagMC uses 5 (1 for tilt and 4×1 for pan).

PostureMC



- A “posture” specifies the states of all the effectors.
- Offers direct access to all OutputCmds.
- Can load/save from a file.
- Interface to kinematics engine.