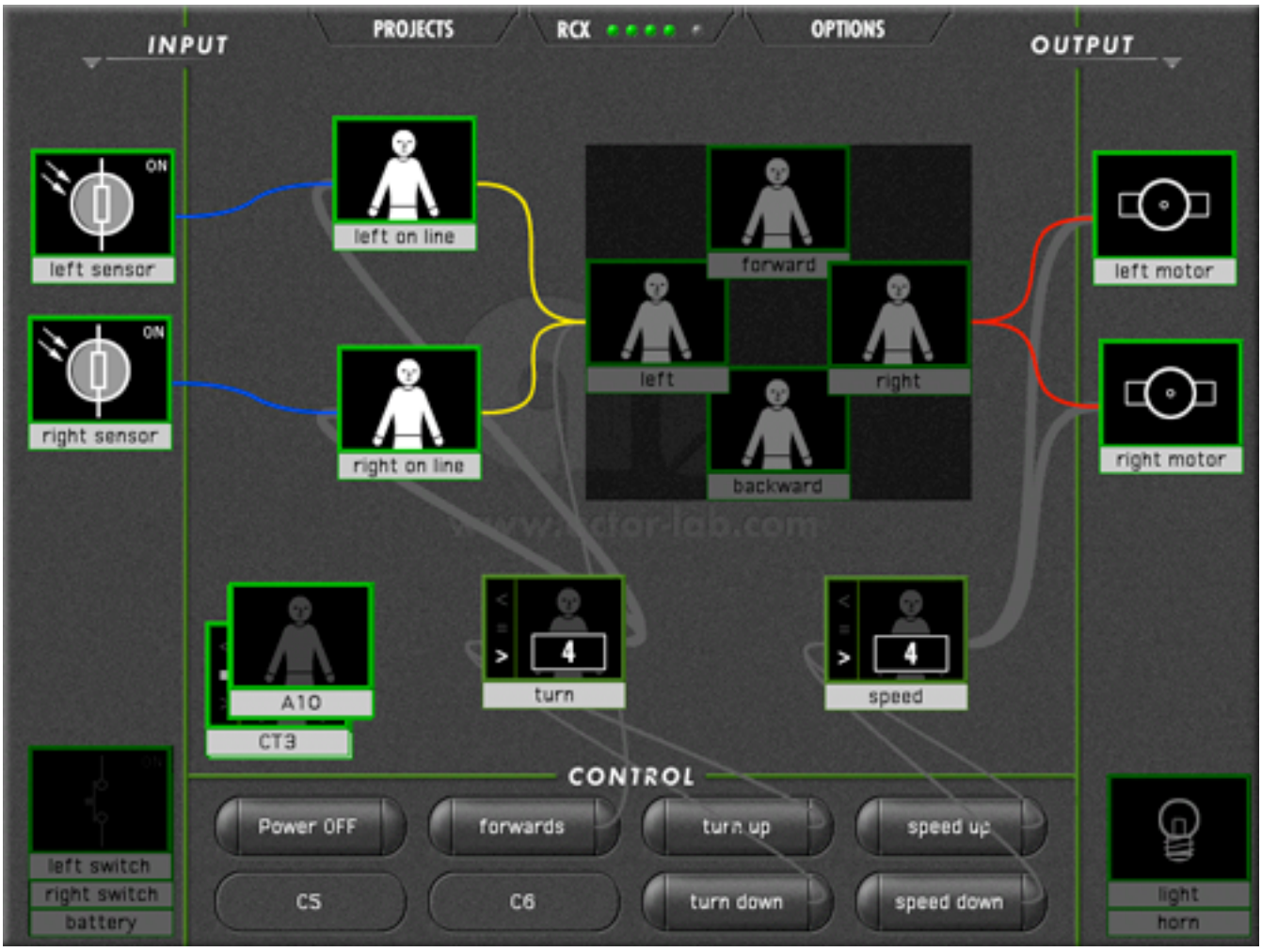
actor-lab

User Manual

Peter Whalley & Ben Hawkridge

KMi, The Open University



Disclaimer

Actor-lab is made freely available for any non-commercial use. The software and documentation are the copyright of the Open University, but may be copied and distributed freely provided that no change is made to them.

No fitness for any purpose is claimed for *actor-lab* and no responsibility is accepted for its use.

No guarantee of any support, explanation or guidance as to the use of *actor-lab* is given.

LEGO, *RCX*, and *RoboLab* are trademarks/copyright the Lego Company, *RoboLab* was developed by Tufts University.

No connection of any kind with either the Lego Company or Tufts University is implied or claimed for *actor-lab* or its authors.

Contents

1. Introduction

Overview

Actor-lab was designed to make the teaching of control technology possible in classroom settings where, after a general introduction by the teacher, year 5-7 children are able to work by themselves in small groups with only peer support. The language and the interface were designed to represent the notion of **input-process-output** that is central to basic robotics and control technology (BECTA, 2004). *Actor-lab* is a control language based on the idea of asynchronous message flow, and represents a simplified graphic version of the *actor* programming model. It's operation can be explained using the intuitively accessible model of a cast of actors able to send messages to each other, with the programmer as playwright. Concurrent processing, identified by Papert (1980) as a fundamental feature of control languages, is implicit and a real-time visualisation of process states and message flow is provided by the *actor-lab* system. Serialisation (sequencing of events) is effected by time-based messaging, and suppressive control of objects (task prioritisation) is achieved with object meta-commands.

Background

The theoretical ideas underlying *actor-lab* are derived from the event-driven, message-passing computer languages that began with the *actor*-based languages devised by Hewit (1976), and further developed by Agha (1986). Practical inspiration was provided by *HyperCard* and to some extent by *Lab View*. The first version was developed in *HyperCard* (Whalley, 1992), but the problems associated with having to input scripts via the *HyperCard* editor meant that it was only useable with year 7 children. The current graphic interface avoids any problems with syntax errors, and was developed using *Macro Media Director* by the second author. School-based testing has led to improvements in the graphic interface, particularly with respect to the visualisation of message paths and event/data flow. Also a set of documented examples have been developed and tested with year 5-7 children. These involve controlling (and making safe) a crushing machine, controlling a car park barrier (which involves counting cars in and out), and programming a buggy to move through a maze (and follow lines).

Limitations

A severe limitation of most control languages, particularly those using the *Lego RCX* 'brick', is that once the program is downloaded little or no information is available to help children understand why their programs do not work as they had expected. This makes it difficult to arrange for children to work by themselves, and usually entails providing a great deal of adult support. *Actor-lab* overcomes this problem by maintaining a constant link with the *RCX*. A dynamic interface gives information on the state of the input and output conditions, and of even more importance, the internal processing in terms of message flow. This 'tethered' operation can only be achieved at some cost to the overall response time of the system, but this is of much less importance than the than the ability to work in whole-class setting within the

middle-school. The internal cycle time of the system and the display refresh time is of the order of 0.05 sec. However each input object, which may represent both an analogue device and a switch, takes approximately 0.15 sec. Whilst it would be unwise to attempt fly-by-wire control with *actor-lab*, such benchmark robotic tasks as maze and line-following are easily achievable.

References

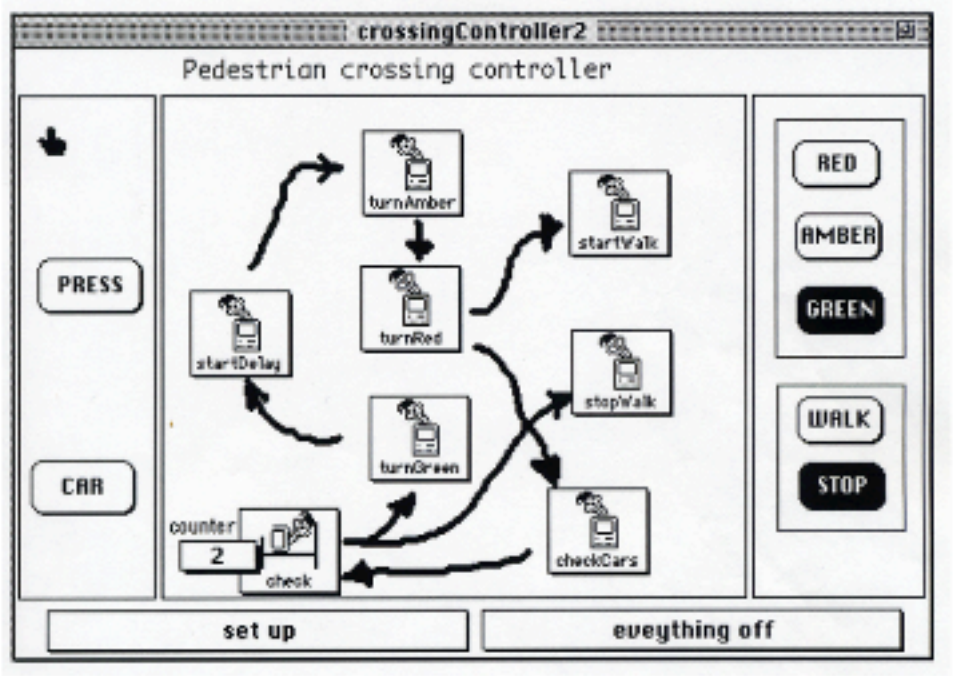
Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*, Artificial Intelligence Series, MIT Press. 1986

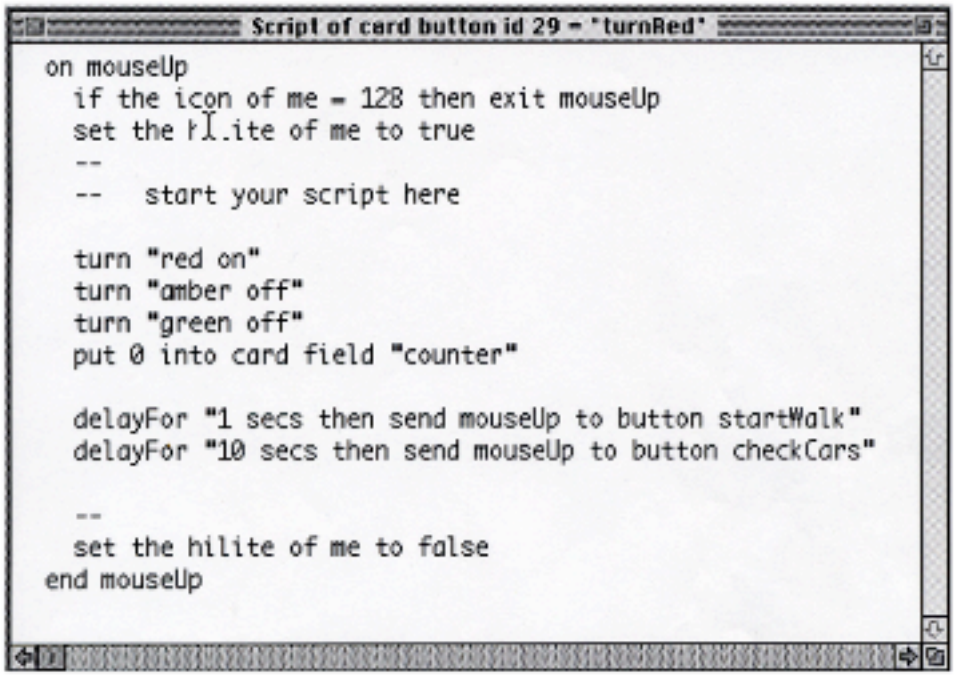
www.becta.org.uk/subsections/foi/documents/technology_and_education_research/control_technology.pdf

Hewitt, C. *Viewing Control Structures as Patterns of Passing Messages*, AI Lab Memo 410, MIT 1976.

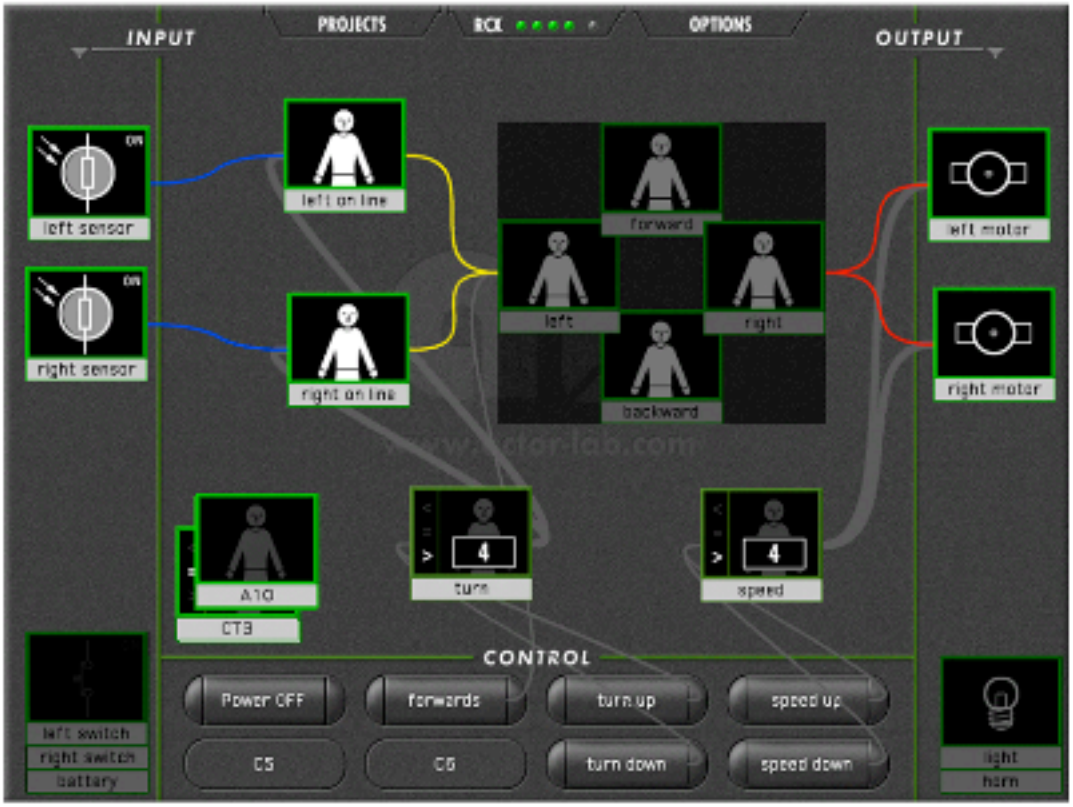
Papert, S. *Mindstorms*, USA: Basic Books, 1980.

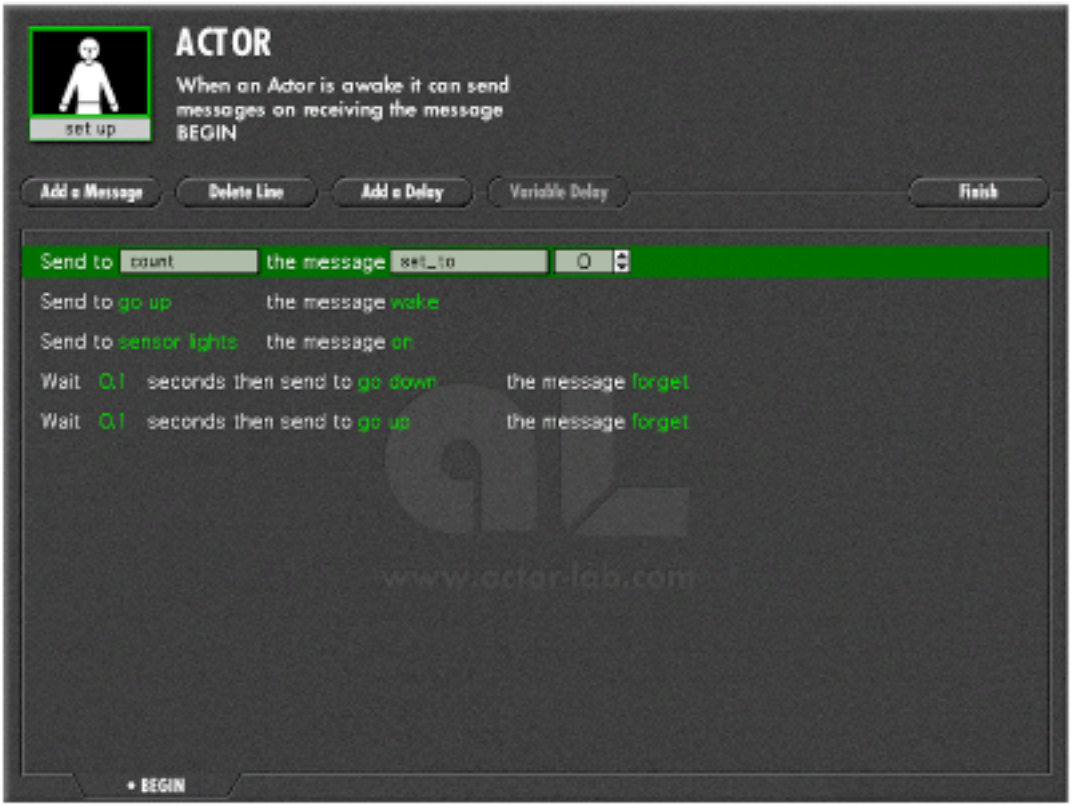
Whalley, P. *Making Control Technology Work in the Classroom*, British Journal of Educational Technology, Vol 23(3), pp 212-221, 1992.





Original interface and editor (1992)





Current interface and editor (2004)

2. How *actor-lab* works

- The key idea to understand is:

input => process => output

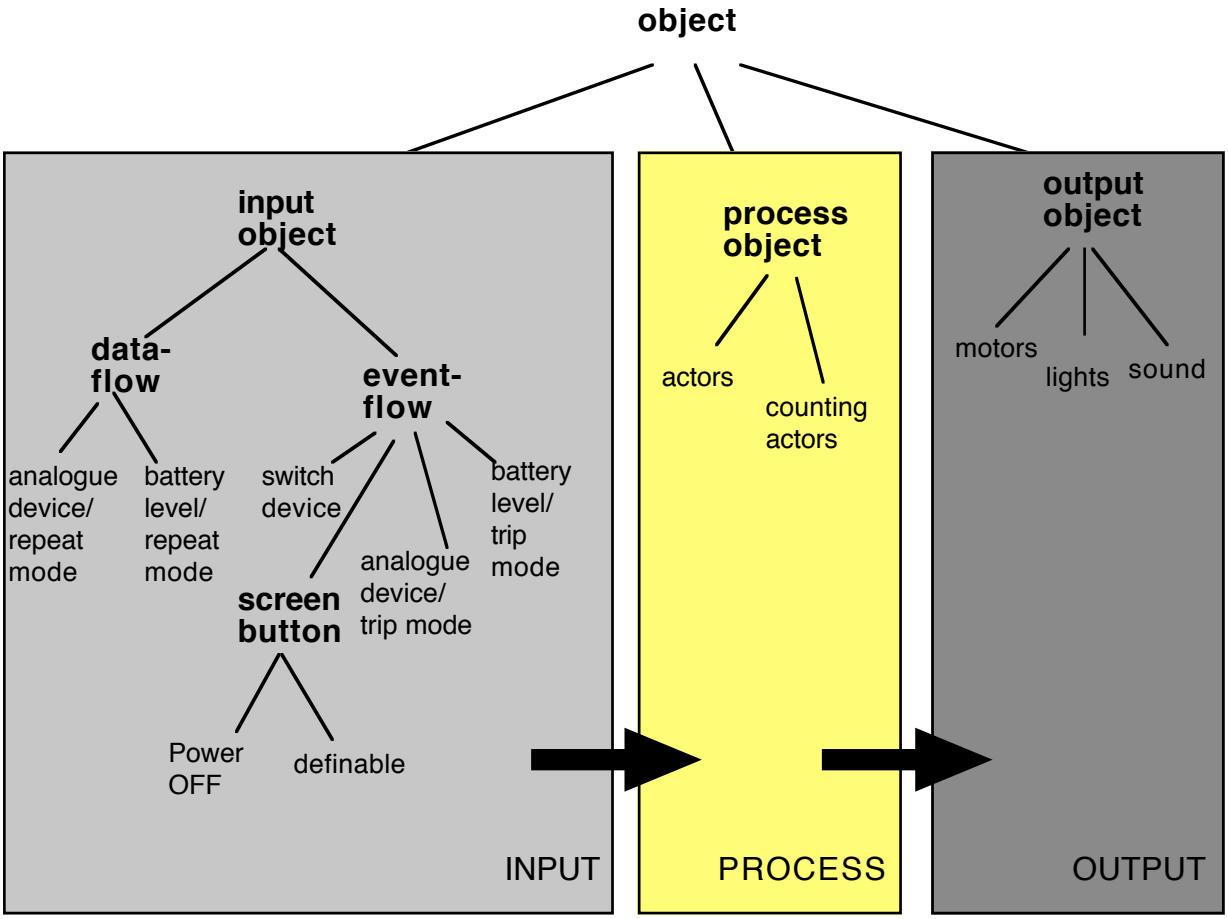


Fig 1. The different types of objects in *actor-lab*

- The basic cycle of *actor-lab* involves real world events being transformed by the INPUT OBJECTS and sent as ***messages,*** or as a flow of data, to PROCESS OBJECTS.

- PROCESS OBJECTS in turn create a flow of **messages** to each other and on to OUTPUT OBJECTS.

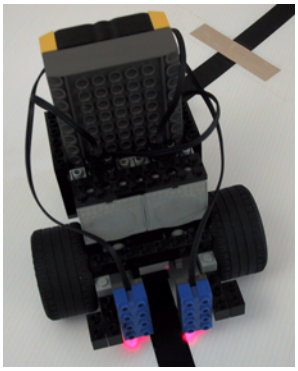
- OUTPUT OBJECTS then bring about real world events.

Important details:

- *actor-lab* works on Mac (OS 9 & OS X) and on PC (Windows 98/2000 & XP)
- *actor-lab* application has to be in a folder with projects, sounds etc.
- **double-click** *actor-lab* application to begin
- to quit application: **Command + Q** on Mac, **Control + Q** on PC
- **click** to open objects
- objects have to be given a unique name to appear in the message list of other objects
- if a name is keyed in that has already been taken, the name will revert to whatever it was before
- if the name of an objects is changed, e.g. to correct it's spelling, any reference to the name in the scripts of other objects is changed automatically
- hold **alt** down to move objects around
- overlapping objects are grouped and only shown as having one input/output path
- hold **alt** down and drag on darkened overlap area to move grouped objects around
- hold **alt** down to drag objects out of a group
- **alt click** on background to dim/brighten paths
- **alt click** to open up screen buttons at the bottom of the display
- serial port selection under OPTIONS (plus advanced mode scripting)
- load/save under PROJECTS
- click on the lights for info/cycle time (improved by disabling inputs)
- the POWER OFF button turns off all outputs, and sends a *forget* message to all actors
- a project file renamed as **default.actl** and put in the **default** folder will override the default 'new' configuration
- hold the **alt** key down as you click on NEW (PROJECT) to be able to configure the input and output objects,
  - to have a switch & light on the same input
  - to use a temperature sensor
  - to monitor the battery voltage
  - change the number of actors/counters.

3. Linking to the *RCX*

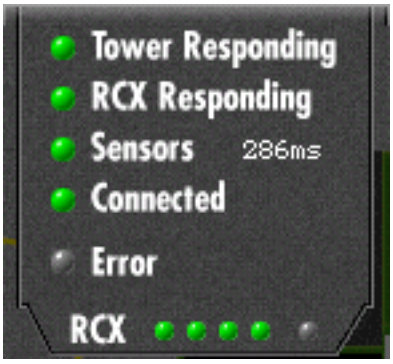
- *actor-lab* requires a serial RCX tower connection (it can be via a USB-serial converter)
- the serial port selection is under OPTIONS
- the RCX brick **must** first have the RoboLab firmware downloaded to it
- *actor-lab* requires **constant communication** with the brick (ie vertical RCX for a buggy)



(see the maze & line buggy manuals)

- 4 green lights showing indicates that you are connected to a brick

click to see this>

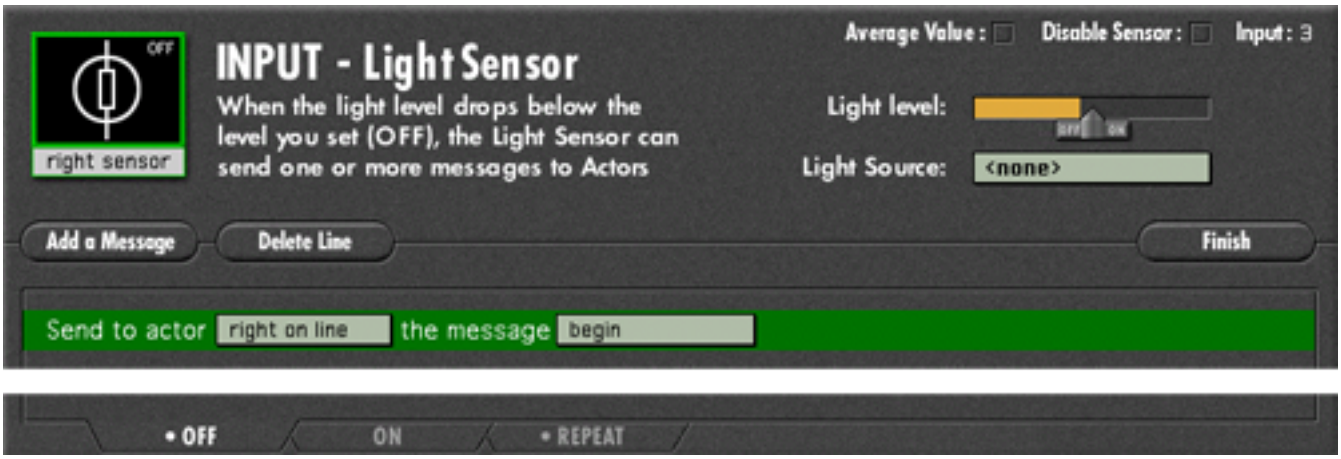


- click on the background to hide the RCX tower extra info panel
- disabling unused inputs is the only way to improve the response time
- *actor-lab* will work in a screen mode when not communicating with a RCX brick (try the 'timer' demo)

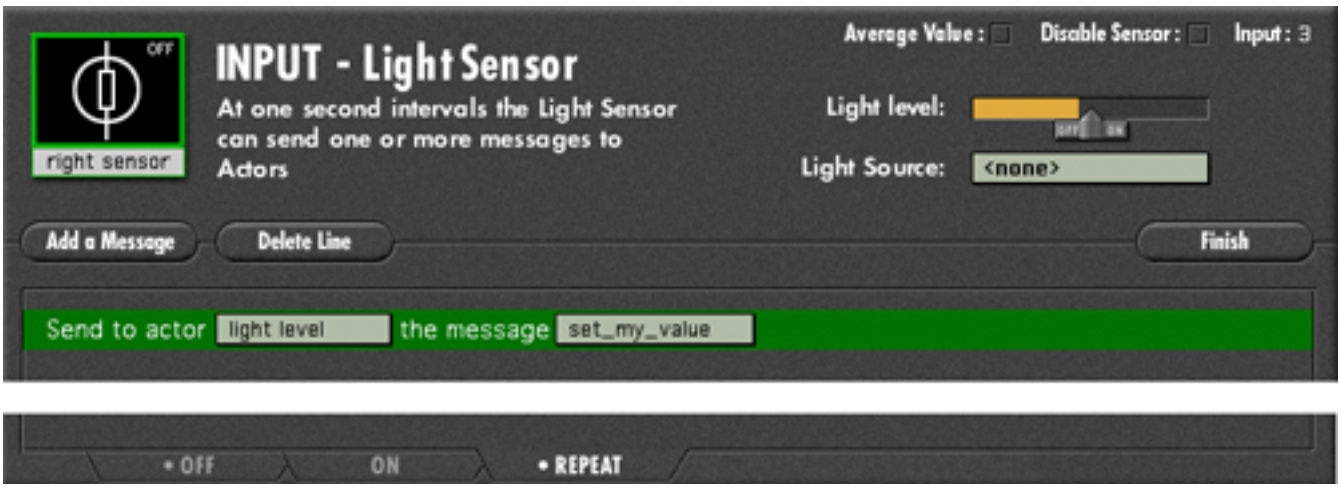
4. Input objects

4.1 analogue devices

Light (and temperature) sensors can send a message (or several messages) when their input level falls below **or** rises above the value set by the slider. Clicking on the input object opens up the edit window and turns on any output light source associated with the sensor. (See the 'crusher' & 'car park' demos.)



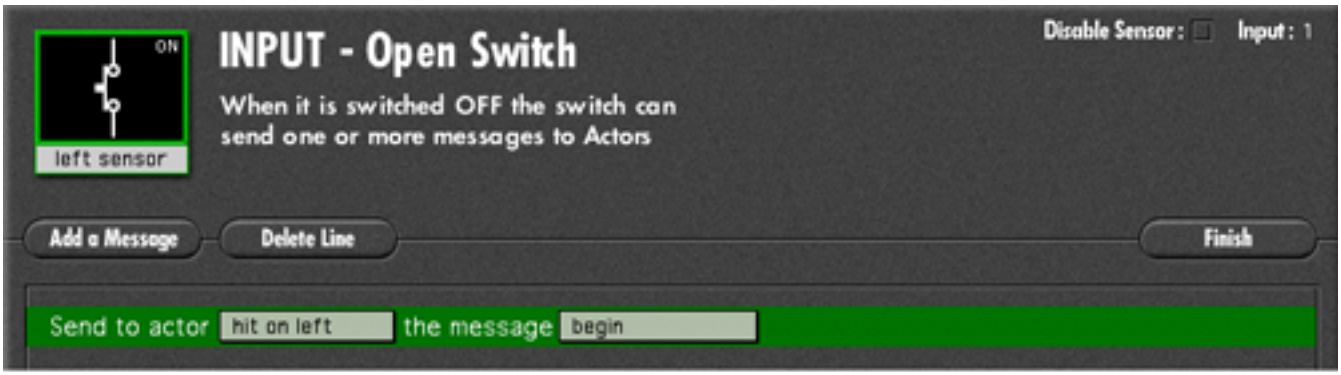
Light (and temperature) sensors can also send a stream of values (and other messages) with a 1 second repeat cycle. (See the 'line buggy' and 'sensor ANDing' demos.)



Disable any unused inputs to improve the response time. The minimum time is 0.15 sec for one input- which could have both a switch and a light sensor attached. The battery level is only sampled every 5 seconds and has little effect on the response time. (The 'increasing' parameter is only useable with high charge rates.)

4.2 switches

Switches can send a message (or messages) as they are pushed in **or** when they are allowed out (in = ON, out = OFF). The icon in the edit window changes to show the current state of the switch.



4.3 buttons

There are 7 user definable screen buttons, plus a 'Power OFF' button. The POWER OFF button turns off all outputs, **and sends a *forget* message to all actors**.

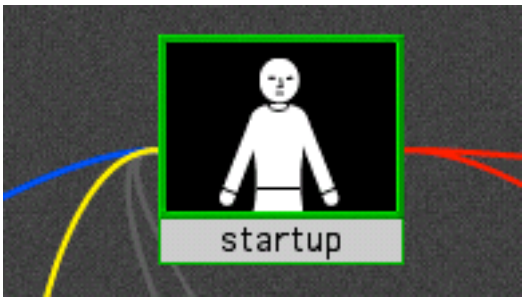
alt click on the definable buttons to open their edit window. The user definable buttons can send one or more messages, but they can not be delayed.

It is possible (but not really advisable) to send messages **to** the screen buttons.

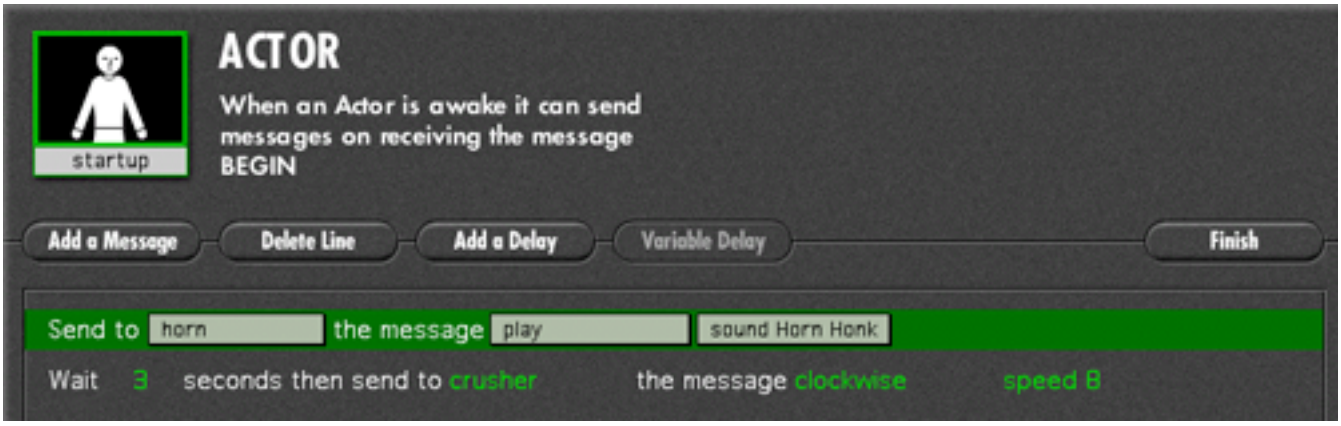
5. Process objects

5.1 *actors*

Understanding how the *actors* operate is the key to working with *actor-lab*. The best metaphor to present to children to describe how they are going to develop their programs is to think of themselves as 'writing a play', giving each *actor* a set of messages (a script) to carry out.



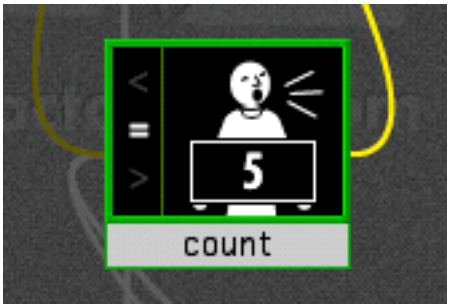
Clicking on an *actor* opens up the edit window that allows you to enter the script. If you hold the **alt** key down whilst dragging on an object you can move it around. Messages from input objects are shown in blue, from other actors in yellow, and from buttons in grey. Messages to output objects are shown in red. When *actor-lab* is working (which is **all the time** that the edit window is not open) you can see the messages being sent as 'particles' moving along the lines.



To begin with it is best to think of *actors* as simply responding to BEGIN messages, which cause them to send BEGIN messages on to other *actors* or specific messages to output objects, e.g. to turn on a light. The only complication at this level is that messages can be sent on **immediately** that the *actor* receives a BEGIN message or **after some delay**. Reasonably complex control problems can be tackled with no more knowledge than this, e.g. Crusher Tasks 1-5 and Car Park 1-2 demos. For more difficult problems it is necessary to understand more about *event sequencing* and also a little about an important issue for control/robotics, *task prioritisation*. These are dealt with in section 7.

5.2 counting actors

Rather than just responding to to BEGIN messages, counting actors respond to changes in their ***value,*** a number between 0 and 9999. They constantly compare their value with a target value, which can either a fixed value or another *counting actor*.



When the value matches (ie it is 'greater than', 'equal to' or 'less than' the target) *counting actors* send messages on, much like other *actors*.

The value that a *counting actor* has is set by messages from *input* objects and other *actors*, using the arithmetic operator messages that can only be sent to *counting actors.*

Another way that a *counting actor* can be used is simply as a value (a variable) in messages sent by other *actors*. The value could be-

- sent to other *counting actors*
- used to set the speed of motors
- used to control the flashing rate of lights
- used to control the time delay of a message

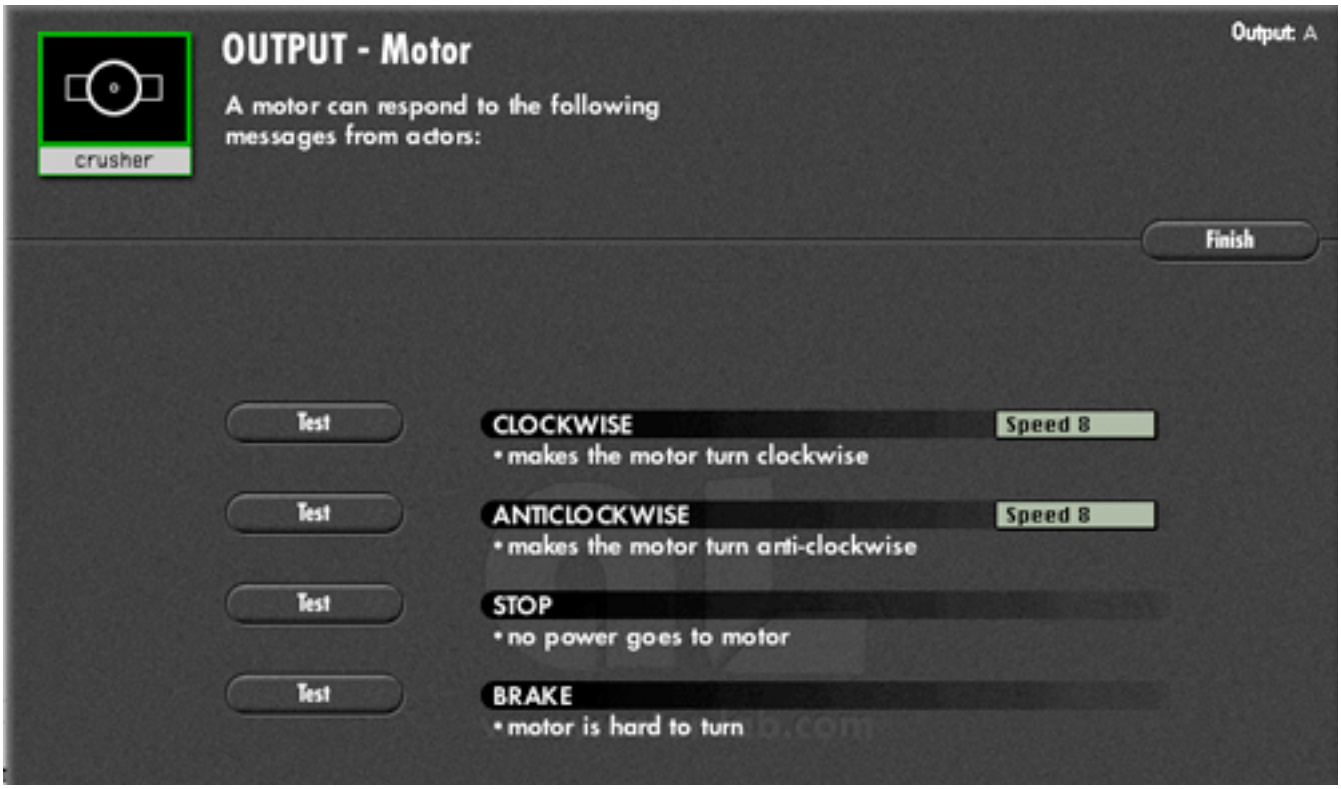
See the 'timer' and 'line-buggy variable' demos for examples of these uses.

The 'sensor ANDing' demo shows the most complex use of a *counting actor*.

6. Output objects

6.1 motors

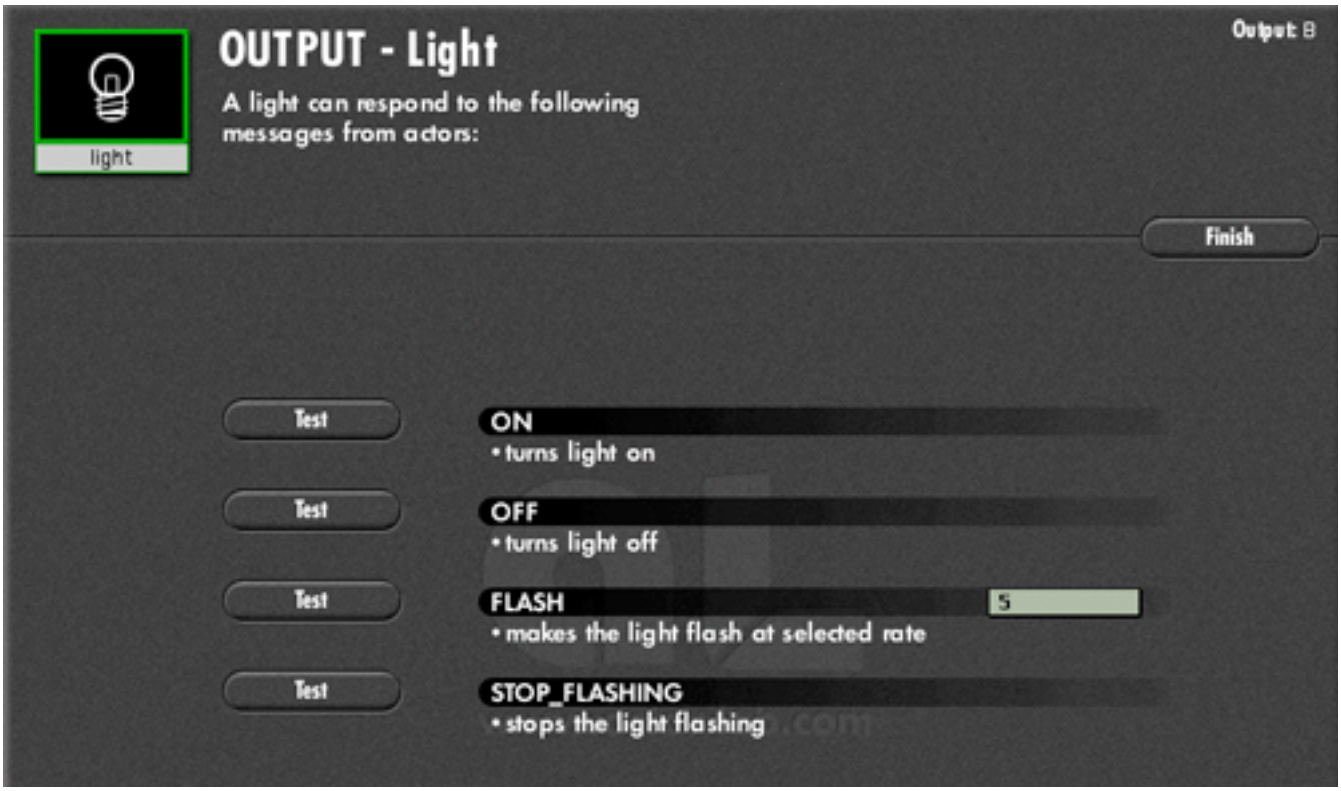
It is usually a good idea to test out motors in order to discover the relationship between clockwise/anticlockwise and up/down, forward/backwards etc.



If any counting actors have been given names, then they appear below the fixed speeds as a possible parameter. (See 'line buggy variable' demo.)

6.2 lights

It is possible to test out light objects by clicking on them.



It is important that the messages be paired correctly, 'stop-flashing' does not cancel 'on', and 'off' does not cancel 'flash'.

The flashing rate can also be assigned to any named *counting actor*.

6.3 sound

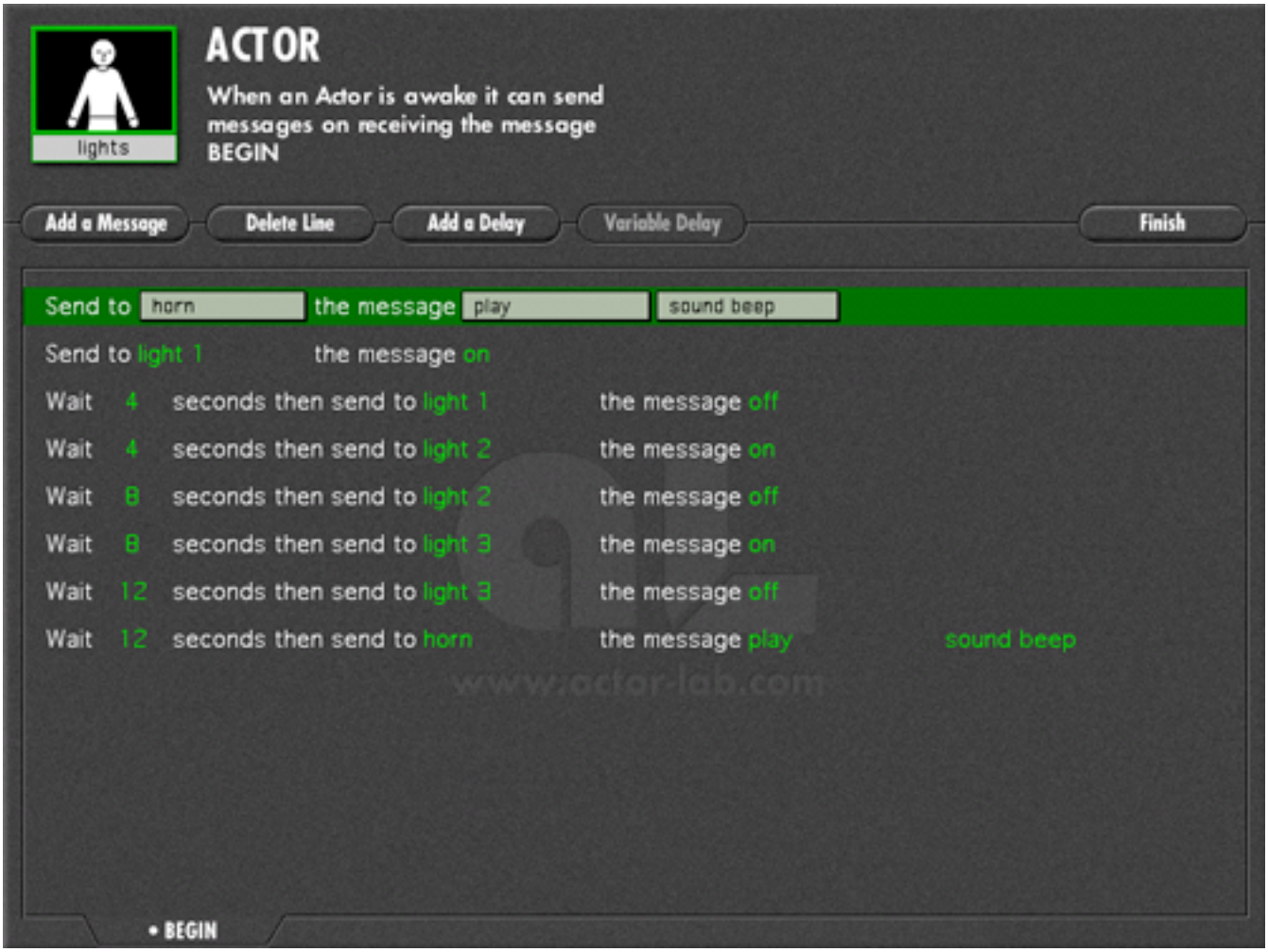
Any **aiff** sound file placed in the Sounds folder will appear in the menu, the 'beep' sound is the current system beep.

7. Process control

7.1 event sequencing

Actor-lab is a time-based language. All processing takes place concurrently (as far as the the user is concerned) as each *actor* has its own process thread. Consequently messages that are not delayed are all sent 'at the same time'. It also means that each *actor* effectively has its own clock, and script messages that are to be delayed are individually time-stamped, with respect to the event that initiated them, and sent at the appropriate time. (Unless meta-commands cause the actor to be disabled at the time that the message should be sent.)

It is easy to create a sequence of timed events in *actor-lab*, but it is very important that users understand that all the delays in a script are with respect to the original BEGIN message. Because the script is of necessity an ordered list (and the editor keeps it in order if delay timings are changed), it is possible to misread the delay times as having a cumulative effect. The '3 lights' demo is quite effective at communicating the correct interpretation. (It is possible that colour banding of 'at the same time' messages might be introduced at some stage.)



Complications (that most users do not need to know about)

An *actor* can be repeatedly fired, even while it has messages that are waiting to be sent. The effect is that any delayed messages would be added to the stack (with a new time-stamp), and any 'immediate' messages would be sent straight away. This is likely to have unwanted consequences, which can only be avoided by controlling the input of messages to the *actor*. See the 'debouncing' effect in the 'timer' demo, and look at the script in the 'SET' actor of 'Car Park 4' for ways to overcome any problems.

The underlying processing is of course not actually concurrent, but has been designed to appear so to users controlling slowly moving models. The underlying cycle time of the system is ≈ 0.05 sec, and each *actor* is then processed in a random order. It is possible, but difficult, to achieve indeterministic results if messages are passed between actors with no buffering delay. The 0.1 sec delay was added to solve any problems. The minimum delay used in normal user programming is 1 sec, and so the 'particles' indicating message flow are timed to take ≈ 0.9 sec. Turning on the 'slow messages' condition causes messages to be visible as a 'ripple' through the *actors* (see the cascade demos), but of course makes control of real world events problematic. This condition is useful in debugging some logical problems, particularly those concerned with event prioritising.

== click ==>

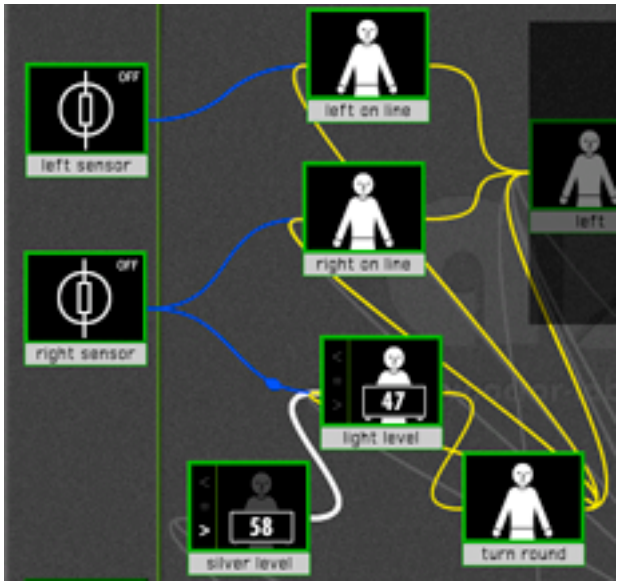
7.2 task prioritisation

Beyond a certain level of problem complexity, it becomes necessary to constrain the concurrent operations. In *procedural* control languages (e.g. *RoboLab, MindScript, NQC*) this is achieved by setting task priority levels or by creating an overriding *subsumption architecture*. In *agent/actor* languages (e.g. *MultiLogo, actor-lab*) task prioritisation is achieved by the 'suppressive control of objects', ie *actors* can send, and be sent, *meta-commands* that change their state. If the 'Allow Sleep/Wake' condition is checked then additional commands are made available (not advisable in the first session with Year 5 children!)

A **SLEEP** message will cause an *actor/counting actor* to become dormant, a condition in which they will **only** respond to a **WAKE** message. Whilst dormant, any delayed messages will not be sent. If the *actor/counting actor* receives a **WAKE** command, it becomes active again, but only on the next process cycle ie ≈0.05 sec later. If **WAKE** and **BEGIN** are to be sent by the same *actor* then the following syntax is necessary:

A **FORGET** message deletes the stack of delayed messages. In the case of a *counting actor*, all three message stacks are deleted.

Examples

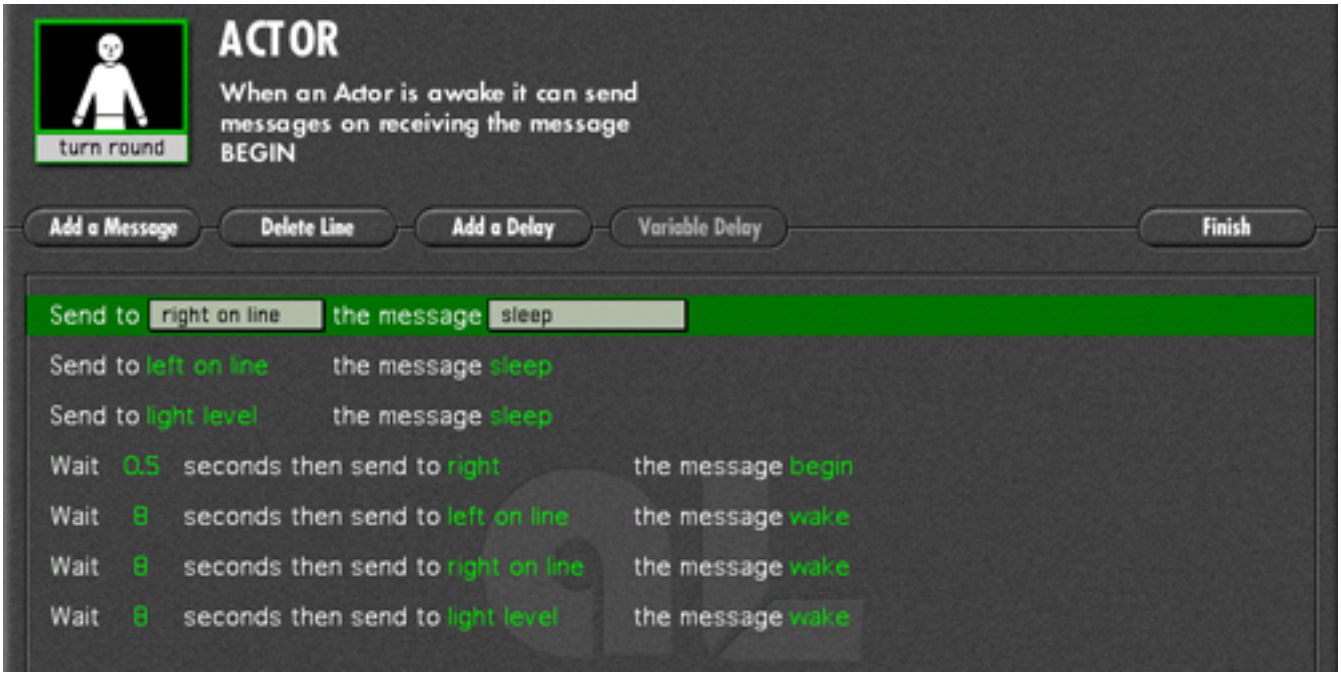


In this example (load: **line buggy foil**) the left and right sensors have their trigger level set so that they send a BEGIN event, to the actors 'left on line' and 'right on line ' respectively, as they move over the black tape. The right sensor is also set up to send a stream of data to the counting actor 'light level'.

The counting actor 'light level' is set up so that when the value of the data stream goes higher than the value in the counter actor "silver level' it will send a BEGIN event to the actor 'turn round'.

The messages in actor 'turn round' make sure that all the actors that might respond to a changing light level are disabled **before** the right turn is initiated. After the sensors have safely passed over the foil and black tape they are enabled again.

(See the web site for video.)



8. Classroom use

Control technology/robotics is a highly motivating and rewarding part of the curriculum for students. Unfortunately it is quite difficult to teach in the 'whole class' environment with normal levels of support and equipment. *Actor-lab* and the exercises described here have been designed to overcome some of these problems.

The ideal way to discover control technology/robotics is to build a model and then learn how to control it by experimenting, and this is quite possible in the context of after-school clubs with extra adults around to help the teacher. *Actor-lab* was designed to be used by a teacher, working with a whole class, and having to rely on the children to help each other to a large extent. (An 'advice' system has been developed to help identify problems, but is not yet ready for release.)

A set of problems with student manuals can be found at:

http://actor-lab.open.ac.uk/

The manuals are initially fairly directive, but become more open ended. They have had two years of trials with year 5-7 students. It is probably best to introduce the key ideas to the whole class, with a system connected to an overhead projector, to get over the key ideas-

- the input => process => output model
- the programmer as playwright, developing *scripts* for a cast of *actors*

The Year 5 topics cover 'controlling machines and making them safe', the Year 6 topics introduce 'machines that react', and the Year 7 topics develop the idea of 'machines that can find their way around'.