

State Machines and Active Object Frameworks

An introduction to the software architecture of real time systems, with illustrative examples from Lego Mindstorms.

Iain McInnes 2004-05-18

Programming Robots is hard

- Build the system from the ground up.
- Likely have poor debug facilities.
- State behaviour becomes complex quickly - humans aren't good at visualising it.
- There are probably several things going on at once
- Real time constraints can be hard to achieve - hard to analyse.
- RTOSs can easily hurt as much as they help: Race conditions, and scheduling problems.

We need all the help we can get.

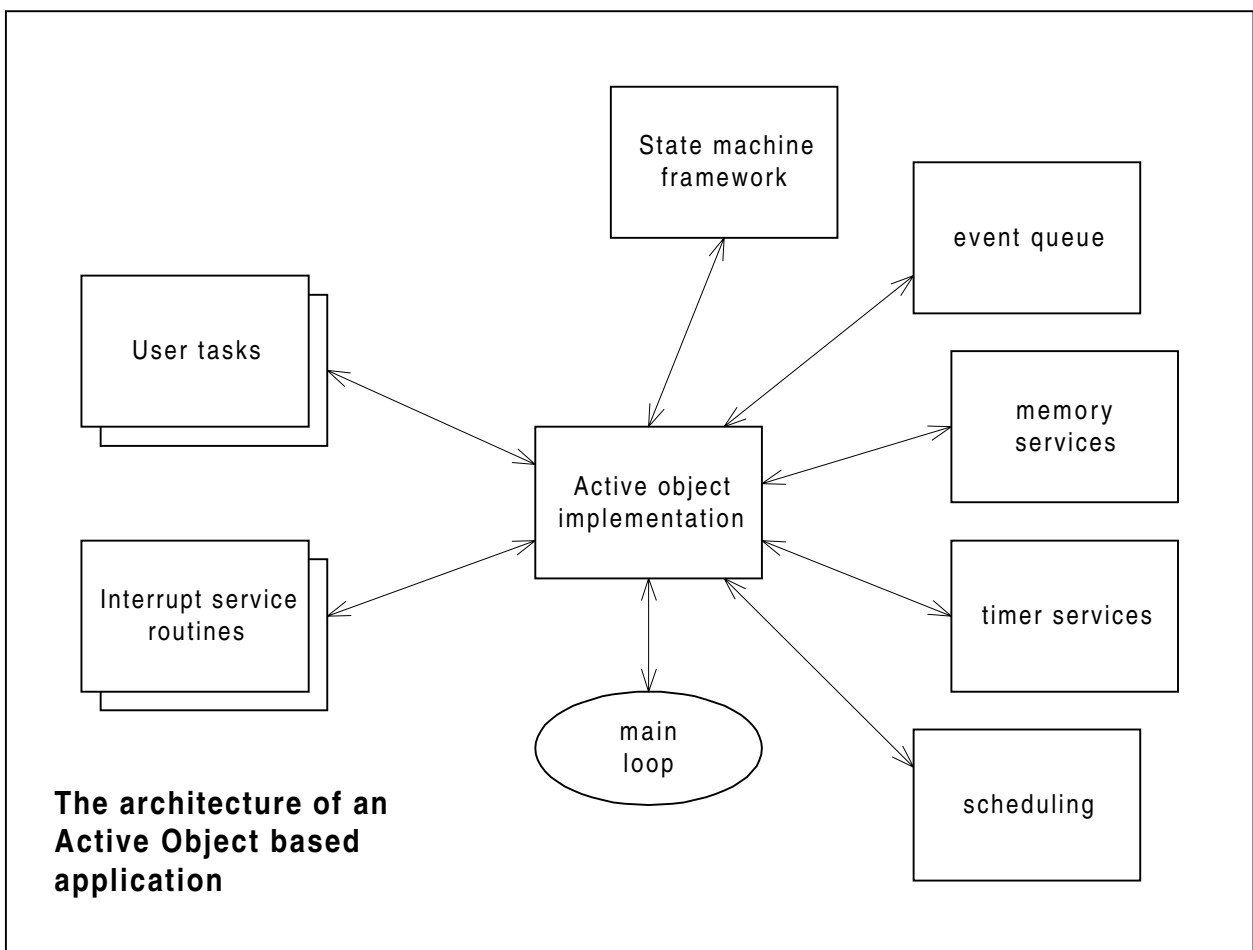
What Must a Decent Real Time Architecture Provide ?

- A platform for application code to run on
- Device drivers
- Timer services
- Task scheduling
- Task communication
- A state machine framework
- Memory services

Snake Oil ?

What if I had a technique that was simple to understand, easy to program, small, efficient, one which hardly anyone knows of - and I'm not selling this information - I'll tell you for free.

That technique is to design around the concept of active objects.



A Round Robin real time architecture

```
while (true)
{
    if (some_set_of_conditions_1  ())
        {
            change_some_flags_1      ();
            generate_some_outputs_1  ();
        }

    if (some_set_of_conditions_2  ())
        {
            change_some_flags_2      ();
            generate_some_outputs_2  ();
        }

    if (some_set_of_conditions_3  ())
        {
            :
        }

    :
}
```

Advantages of the Round Robin

- For simple problems, it doesn't get any easier than this.

Problems With the Round Robin

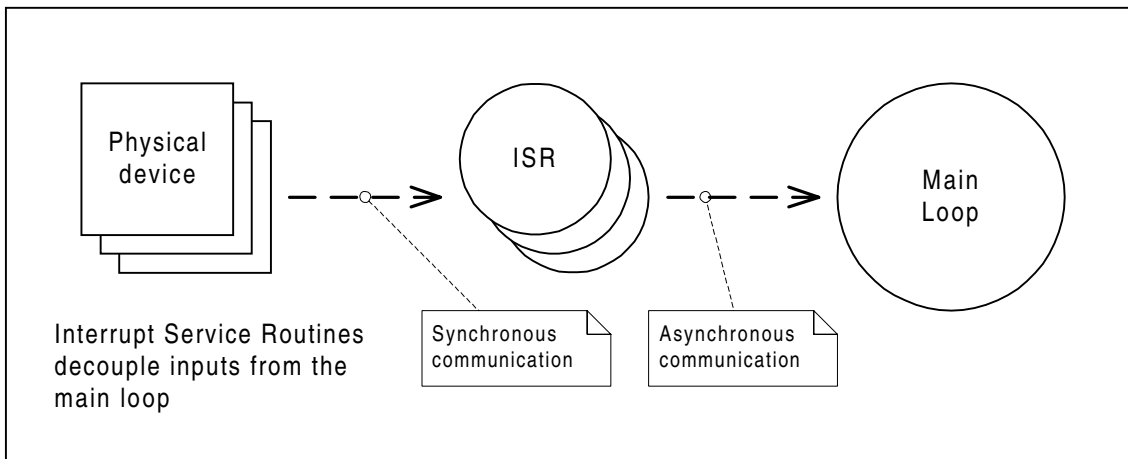
- The worst case latency is the time of the longest transit through the entire loop (there are no individual tasks)
- No distinction between input flags and state.
- Ad-hoc flag variables do not scale well: Humans are poor at dealing with combinations -
 - Do the conditional statements deal with all of the valid combinations of input ?
 - Are they all consistent ?
 - Are the flags always set properly in response to inputs, and previous state ?

There is no real way to know.

Interrupts can improve latency

- Many latency requirements can be met if the system can respond to external events when they occur.
- An interrupt is like a function call - but you can never predict when it will occur.
- The communication between an ISR and the main program is asynchronous.
- Chaos can occur if the ISR modifies data which is owned by the main program.

Interrupts are a necessary evil in many real time systems.



Active Objects

Active objects address the large scale partitioning and concurrency issues of a real time system.

Active objects are like ideal employees in a well run business

- They have a well defined job to do
- They communicate as required to get the job done.
- Each chooses what to do, how to do it, when to do it on their own schedule.

Active objects kiwi blokes of software world:

- They have their own way of doing things
- They communicate only as much as required, as infrequently and as tersely as possible.
- They do things on their own time.
- They are strong.

What is an active object

- A key structural abstraction in a real time system.
- A program entity which can communicate asynchronously, and which has Run To Completion semantics.
- A state machine with an event queue
- The root of a thread ?
- A task

Active objects communicate asynchronously via events or messages. A message is an event with data. They do not share data in any other way. The only time that they block is while waiting for an event. They can receive any event in any order (as defined by the semantics of the event model). Active objects are typically implemented as a state machine.

- Active objects are very modular.
- Active objects are very easy to schedule.
- Active objects do not suffer from race conditions.

I will present

- A set of design patterns - policies
- Implementation Techniques
- A code framework in C implementing these services.
- Comparison with other techniques: - when do you need simpler; when do you need more complex ?

Contents

- 1) Introduce active object based design - what does the overall design and the application code look like. Notation. A line follower example.
- 2) How to implement state machines and event queues.
- 3) Scheduling; pre-emption; RTOSs;
- 4) Further techniques - Hierarchical state machines, timers, memory pools

A Simple Example: A line follower

The robot has two motors; Motor A drives the left wheel, Motor C drives the right.

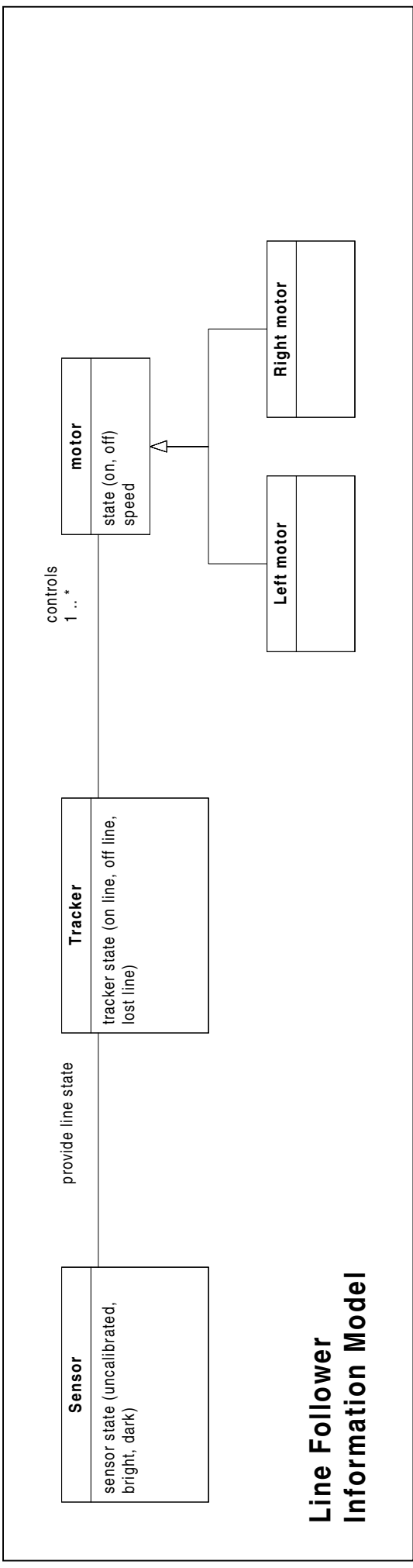
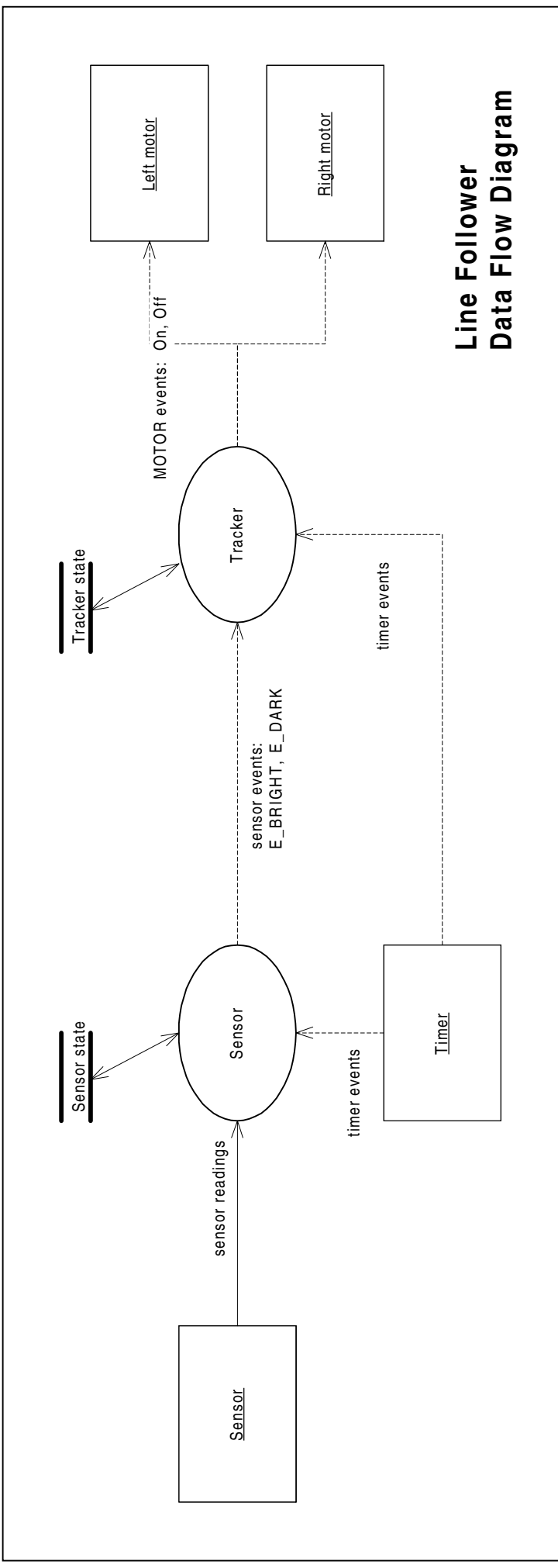
The robot has one optical sensor. The robot is initially positioned so that if it goes forward, it will encounter the line that it should follow. It should calibrate its light and dark thresholds when it encounters the line.

The robot should endeavour to follow the right hand edge of the line:

If the sensor records dark, then turn to the right.

If the sensor records light, then turn to the left.

After being off the line for one second, then back up and attempt to find it again.

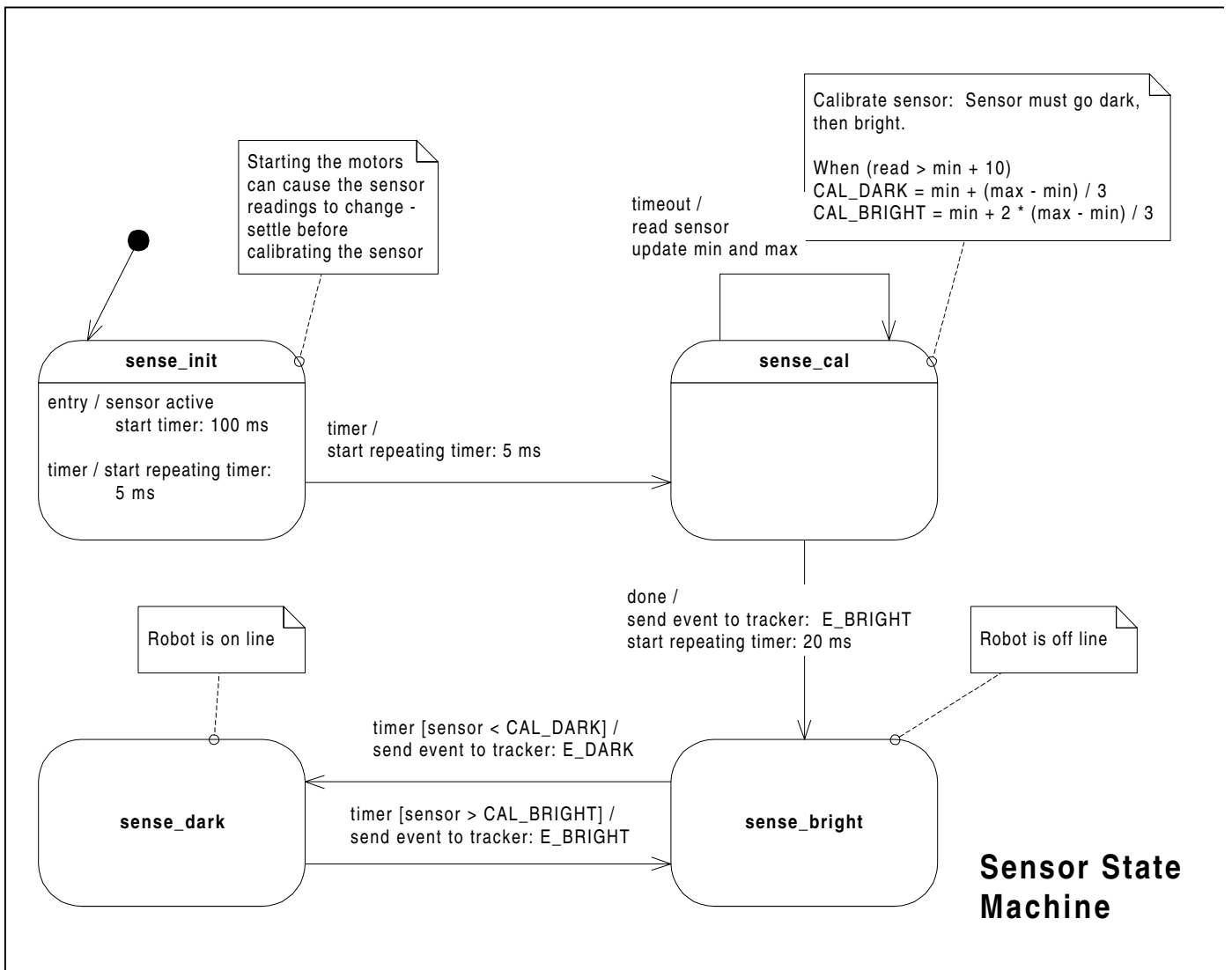


Data Flow Diagram

- Analysis and modelling technique using DFDs and state machines pioneered by Ward and Mellor in mid 1980's
- One top level diagram showing the active objects is a very good way of expressing the communication model of the application.
- The entities are stateful.
- The arcs represent information flow.
- Is less useful as a modelling / design technique - you have to make too many decisions up front, when the full behaviour of the application is not well understood.

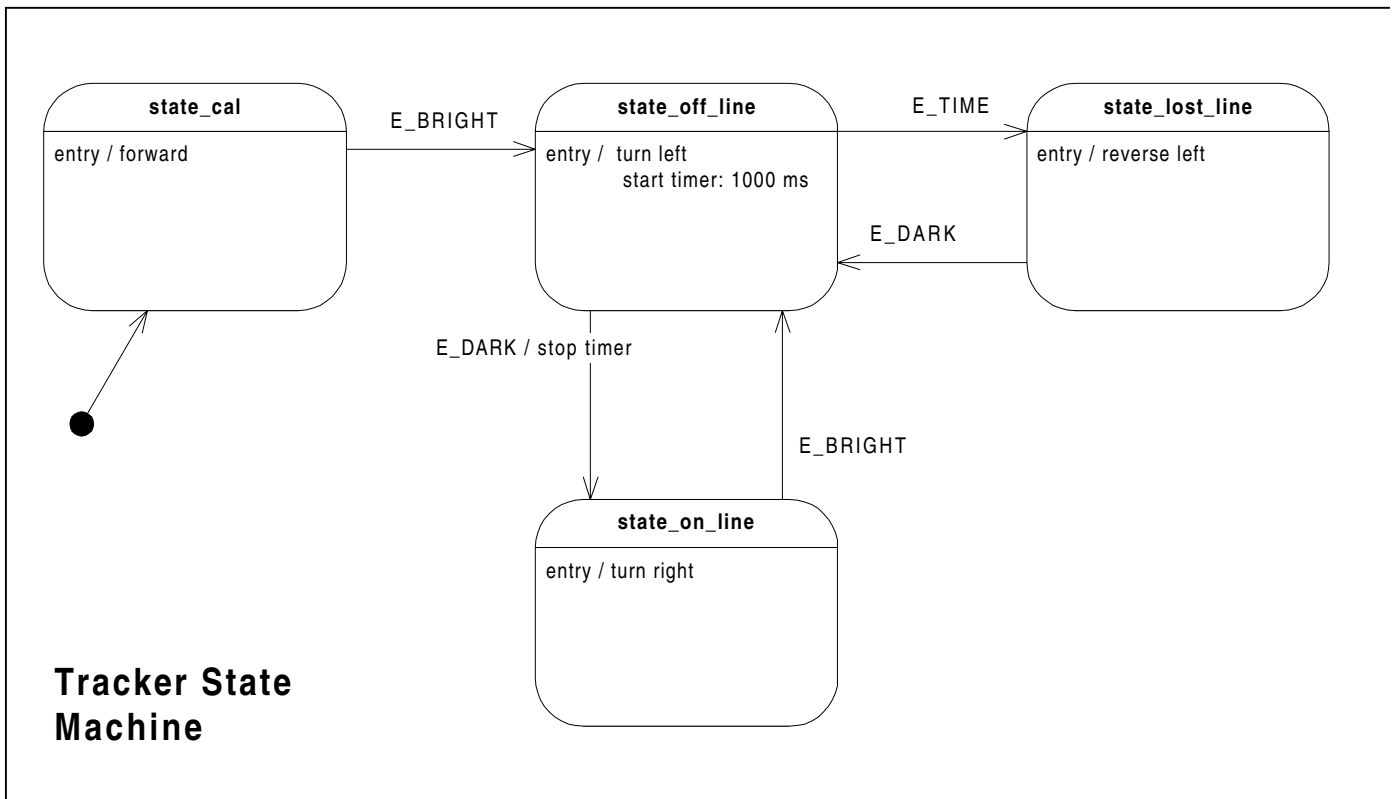
(Object) Information Model

- Does not distinguish the system boundary: Is a model of "the real world".
- Models all of the data (state) in the application.
- Does not care about synchronisation boundaries - they come later.
- The entities are information (objects)
- The relationships represent communication - but only in an abstract sense. They represent knowledge and services.
- The relationships contain multiplicity information.
- This notation is that of a database schema.
- Is a powerful means of expressing, and therefore thinking about systems.



```

void sense_bright (sm_t * psm, event_t e)
{
    switch (e)
    {
        case E_ENTRY:
        {
            // cputs ("bri");
            break;
        }
        case E_TIME:
        {
            if (LIGHT (LIGHTSENS) < CAL_DARK)
            {
                ao_send_event (p_line_track, E_DARK);
                sm_goto (psm, SENSE_DARK);
            }
            break;
        }
    }
}
  
```



```

/* off line: turn to the left */
void state_off_line (sm_t * psm, event_t e)
{
  switch (e)
  {
    case E_ENTRY: m_turn_left();
                  ao_timer_start (p_line_track, 1000);
                  cputs ("off");          break;

    case E_DARK : ao_timer_stop (p_line_track);
                  sm_goto (psm, S_ON_LINE ); break;

    case E_TIME : sm_goto (psm, S_LOST_LINE); break;
  }
}
  
```

State Machine Design Idioms

- The notation is standard UML statechart notation.
- The initial transition initialises the state machine (initial transition may have actions).
- Use entry actions in preference to transition actions - SMs are less cluttered; Moore state machines have output as property of state.
- Can also specify exit functions - less often needed.
- 'Standard' events: E_INIT, E_ENTRY, E_EXIT, E_TIME
- Events that do not cause transitions are shown inside the state.
- Notation for transitions is event / [guard] actions, eg:
timer [sensor < CAL_DARK] /
send event to tracker: E_DARK
- State machines can be hierarchical (not shown)
- The UML allows other (complicated) mechanisms - orthogonal regions, state history.
- Events can be any lightweight type eg enum { . . . } that is easily copied.

Observations about the State diagrams and the code

- The diagrams closely match the code (Modelling tools exploit this property).
- The code is simple and highly structured.
- The diagrams form a specification.
- The distance between the specification and the machine code is small.
- State machine based code can be fast and small.
- The framework provides the actual state machine object - the user code fleshes out the skeleton.
- The code depends on many services:

Events:	<code>E_ENTRY, E_TIME</code>
State transitions:	<code>sm_goto (psm, SENSE_DARK);</code>
Object communication:	<code>ao_send_event (p_line_track, E_DARK);</code>
Timer services:	<code>ao_timer_start (p_line_track, 1000);</code> <code>ao_timer_stop (p_line_track);</code>
Output functions:	<code>m_turn_left();</code>

The contract between the user code and SM implementation:

User Code Provides (defines)

Events related to the application

Event handlers (one for each state) with the signature:

```
void handler (sm_t * psm, event_t e);
```

A table of pointers to the event handlers - allows the state to be represented as an enumeration.

Enumerates the states.

State Machine Engine Provides:

Engine remembers the current state - is an object which can be instantiated.

Predefined events, `E_INIT, E_ENTRY, E_EXIT, E_TIME`

Initialisation: Engine enters the first state in the handler table.

Implements the go-to-state operation - `sm_goto ()`

Invoke entry actions - call state handler function with entry event.

Represents the state machine to the rest of the framework.

Application Code

```
/*
*****
/*
/*      A simple line tracker demo.
/*
/*
/*      Lines are assumed to curve to the right, and initially encountered
/*      from the right (for example, from the inside of the test pad).
/*      Straight line tracking needs to define STRAIGHT_LINE.
/*
/*
/*      Assumes motors on A,C, light sensors on port 2
/*
/*
/*      Iain McInnes 2004-01-02:
/*      Based on a demo program by Markus L. Noga <markus@noga.de>
/*      Modified to use user installed timer, and event queue.
/*
*****
#include <config.h>
#if defined(CONF_DSSENSOR) && defined(CONF_DMOTOR)

#include <conio.h>      // display functions
#include <unistd.h>
#include <tm.h>         // task manager

#include <dsensor.h>   // sensor routines
#include <dmotor.h>    // motor routines

#include <tick.h>      // OS timing
#include "equeue.h"    // Event queues

#include "sm.h"        // State machine engine
#include "actobj.h"    // Active objects

/*-----*/

#define LIGHTSENS      SENSOR_1

#define NORMAL_SPEED (MAX_SPEED)
#define TURN_SPEED    (MAX_SPEED)

/*
Constructopaedia line follower: A motor drives left wheels,
C motor drives right wheels
*/
void m_turn_left (void)
{
    motor_a_speed (TURN_SPEED);    motor_c_speed (TURN_SPEED);
    motor_a_dir   (brake);         motor_c_dir   (fwd);
}
void m_turn_right (void)
{
    motor_a_speed (TURN_SPEED);    motor_c_speed (TURN_SPEED);
    motor_a_dir   (fwd);           motor_c_dir   (brake);
}
void m_forward (void)
{
    motor_a_speed (NORMAL_SPEED);  motor_c_speed (NORMAL_SPEED);
    motor_a_dir   (fwd);           motor_c_dir   (fwd);
}
void m_reverse_left (void)
{
    motor_a_speed (TURN_SPEED);    motor_c_speed (TURN_SPEED);

```

```

    motor_a_dir    (brake);          motor_c_dir    (rev);
}
void m_stop      (void)
{
    motor_a_dir    (brake);          motor_c_dir    (brake);
}

/* Calibrated sensor thresholds */
static unsigned short CAL_DARK, CAL_BRIGHT;

/* Line tracker events: Sensor needs to know these */
enum { E_DARK = E_APP, E_BRIGHT };

/*-----*/
/* Active objects */

static active_object_t * p_line_det;    /* Detect on or off line */
static active_object_t * p_line_track;  /* Track the line */

/*-----*/
/*
Sensor Handler: keep track of whether the sensor input is
bright or dark.
- When cross DARK_THRESH from bright to dark, send an E_DARK event.
- When cross BRIGHT_THRESH from dark to bright, send an E_BRIGHT event.

A more sophisticated handler could do more signal conditioning.

Creating an active object:
1) define the input events. Assign the first to E_APP
2) Declare the state handler functions - one per state.
The first should be an init function
3) Create and initialise a state table containing pointers to the handlers
4) Declare an enumeration for the states that maps on to the state table.
5) Declare any auxiliary variables that will be used by the state machine.
The first (init) handler function should initialise them.
6) Define the handler functions
*/

fsm_handler sense_init;    /* Set up and settle */
fsm_handler sense_cal;    /* find and recognise line; calibrate thresholds */
fsm_handler sense_dark;   /* Detect off->on line */
fsm_handler sense_bright; /* Detect on-> off line */

/* Line follower states */
static fsm_handler * Sensor_states []
= { sense_init, sense_cal, sense_dark, sense_bright };
enum { SENSE_INIT, SENSE_CAL, SENSE_DARK, SENSE_BRIGHT };

static unsigned short sensor_min = 10000;
static unsigned short sensor_max = 0;

/* Uncalibrated; Activate sensor and settle */
void sense_init (sm_t * psm, event_t e)
{
    switch (e)
    {
        /* Make sensor active, and settle 100 ms */
        case E_ENTRY:
            ds_active (&LIGHTSENS);
            ao_timer_start (p_line_det, 100);
            break;

        case E_TIME:

```

```

        ao_timer_repeat (p_line_det, 5); /* sample every 5 ms */
        sm_goto (psm, SENSE_CAL);
        break;
    }
}

/*
go forward until get dark, and then bright. record the extrema.
CAL_DARK  = light_min + (light_max - light_min) / 3.
CAL_BRIGHT = light_min + 2 *(light_max - light_min) / 3
*/
void sense_cal (sm_t * psm, event_t e)
{
    unsigned short read = LIGHT (LIGHTSENS);

    cputw (read);
    sensor_min = (read < sensor_min) ? read : sensor_min;
    sensor_max = (read > sensor_max) ? read : sensor_max;

    if (read > sensor_min + 5) /* have gone past darkest */
    {
        CAL_DARK  = sensor_min + (sensor_max - sensor_min) / 3;
        CAL_BRIGHT = sensor_min + 2 *(sensor_max - sensor_min) / 3;
        ao_send_event (p_line_track, E_BRIGHT); /* tell line tracker */
        sm_goto (psm, SENSE_BRIGHT);
        ao_timer_repeat (p_line_det, 20);      /* sample every 20 ms */
    }
}

void sense_dark (sm_t * psm, event_t e)
{
    switch (e)
    {
        case E_ENTRY:
            {
//                cputs ("dark");
                break;
            }
        case E_TIME:
            {
                if (LIGHT (LIGHTSENS) > CAL_BRIGHT)
                {
                    ao_send_event (p_line_track, E_BRIGHT);
                    sm_goto (psm, SENSE_BRIGHT);
                }
                break;
            }
    }
}

void sense_bright (sm_t * psm, event_t e)
{
    switch (e)
    {
        case E_ENTRY:
            {
//                cputs ("bri");
                break;
            }
        case E_TIME:
            {
                if (LIGHT (LIGHTSENS) < CAL_DARK)
                {
                    ao_send_event (p_line_track, E_DARK);
                }
            }
    }
}

```

```

                sm_goto (psm, SENSE_DARK);
            }
        }
        break;
    }
}

/*=====*/
/* Line Tracker state machine */

fsm_handler state_cal;
fsm_handler state_on_line;
fsm_handler state_off_line;
fsm_handler state_lost_line;

/* States */
/* Look up table for handlers: these must match the enum */
static fsm_handler * Line_states []
= { state_cal, state_on_line, state_off_line, state_lost_line };
enum { S_CAL, S_ON_LINE, S_OFF_LINE, S_LOST_LINE };

/*-----*/
/*
calibrating: go forward until sensor can recognise line and calibrate.
Will get E_BRIGHT msg when success.
*/
void state_cal (sm_t * psm, event_t e)
{
    switch (e)
    {
        case E_ENTRY : m_forward ();           cputs ("cal");           break;
        case E_BRIGHT: sm_goto (psm, S_OFF_LINE); break;
    }
}

/* On the line: turn to the right */
void state_on_line (sm_t * psm, event_t e)
{
    switch (e)
    {
        case E_ENTRY : m_turn_right ();       cputs ("on");           break;
        case E_BRIGHT: sm_goto (psm, S_OFF_LINE); break;
    }
}

/* off line: turn to the left */
void state_off_line (sm_t * psm, event_t e)
{
    switch (e)
    {
        case E_ENTRY: m_turn_left();
                    ao_timer_start (p_line_track, 1000);
                    cputs ("off");           break;

        case E_DARK : ao_timer_stop (p_line_track);
                    sm_goto (psm, S_ON_LINE ); break;

        case E_TIME : sm_goto (psm, S_LOST_LINE); break;
    }
}

/* Have lost the line: back up until find it */
void state_lost_line (sm_t * psm, event_t e)
{

```

```

switch (e)
{
    case E_ENTRY: m_reverse_left ();      cputs ("lost");      break;
    case E_DARK :      sm_goto(psm, S_ON_LINE);      break;
}
}

/*-----*/
/* Shut everything down: turn off sensor, stop timers, stop motors */

void linetrack_exit (void)
{
//    sys_timer_stop (&Sensor_timer);  !!! figure out a solution to this
    m_stop ();
    ds_passive (&LIGHTSENS);
}

/*-----*/
/*
Create a list of active objects, and put tracker and sensor objects on it
Start the objects (put them into their initial states)
Then set them running
*/

/* Event queue size; must be a power of 2; Allocate one more than needed */
#define EQ_SIZE 0x4

int main (void)
{
    active_object_t * p_objects = 0;

    p_line_track = ao_new_fsm (EQ_SIZE, Line_states );
    p_line_det    = ao_new_fsm (EQ_SIZE, Sensor_states);

    ao_add_front (&p_objects, p_line_track);
    ao_add_front (&p_objects, p_line_det);

    ao_start_all (p_objects);
    ao_run      (p_objects);

    linetrack_exit ();
    ao_delete_all (p_objects);
    return 0;
}

/*-----*/
#else
#warning linetrack.c requires CONF_DSENSOR && CONF_DMOTOR
#warning linetrack demo will do nothing
int main(int argc, char *argv[]) {
    return 0;
}
#endif // defined(CONF_DSENSOR) && defined(CONF_DMOTOR)
/*-----*/

```