The LegOS embedded operating system - Part 1

Pramode C.E

23rd July 2002

Abstract

The Lego Mindstorm Robotic Invention System is one of the most fascinating technological innovations in recent years. This document is a tutorial introduction to a reasonably powerful embedded operating system called LegOS which makes Lego coding lots of fun for the serious Linux/C hacker. Through some very simple (and interesting!) programming examples, I demonstrate how Operating System concepts like concurrency and multithreading are employed to control real world mechanical systems. Lego robotic systems are used to teach programming, artificial intelligence and control theory in some of the best universities in the world - including our own IIT's (IIT Kanpur is currently coordinating an interesting project called BRICS). Try searching the web for 'Lego Mindstorm'.

1 What is Lego?

Lego is the world's largest toy manufacturing company. They make 'building blocks' using which kids construct things out of their imagination - the sheer number, variety and ingenious design of the blocks makes it possible to build virtually anything we can think of.

1.1 What is a Lego Mindstorm?

Building 'intelligent robots' is one of the most fascinating of all human endeavours - it is a result of our desire to embed a part of ourselves in inanimate objects and breathe 'life' into them. Robot building was confined to university research departments and big industrial houses, as few individuals had the resources to design and build their own systems.

Lego, together with researchers at MIT's Media Lab, changed all that in the year 1998 with the introduction of their 'programmable brick'. The 'programmable brick' is a small Hitachi microcontroller with 3 motor outputs and 3 sensor inputs. This, combined with standard Lego building blocks and additional parts like wheels, pulleys, gears etc (all made of plastic) can be used to build fascinating programmable robotic systems. The user builds the robot, writes a program on the PC and downloads it onto the 'brick' - the robot now becomes fully autonomous and moves and acts according to the program stored in the brick.

2 A Brief description of the Programmable Brick

The Programmable Brick, which forms the heart of the robotic system, is nothing but a Hitachi microcontroller and 32KB of RAM, with some motor driver and sensor/IR circuits, packed in a small yellow coloured Lego block. Programs and data are stored on the RAM. When the brick is powered up, it executes code stored in about 16K of ROM. This code either transfers control to an operating system (called 'firmware' in Lego Literature) loaded in RAM, or, if firmware is not present, it gets ready to download one through an IR link from the PC. The RAM is battery backed - even when the unit is powered off, the battery keeps powering the RAM.

3 LegOS

It is possible to set up a cross development environment under Linux to develop Lego applications. A port of the GNU compiler to the Hitachi controller is available. Once that is installed, you can get a copy of a small embedded operating system called 'LegOS' and compile it. The compiled LegOS can be downloaded to the programmable brick by using the command 'firmdl3' (which comes with the LegOS distribution). Once the OS (firmware) is downloaded, you can write C programs, compile them into .lx files (that's the extension which LegOS uses - the C programs can't be compiled into native Linux format because they are not going to be executed on Linux) and download them onto the Lego brick by using the 'dll' command. There are 8 slots available for storing programs on the Lego brick - using an option to the 'dll' command, you can store your program in whichever slot you like. Execution of the program is done by first using the 'Pgm' button on the Lego brick to choose a program and then pressing the 'Run' button.

LegOS supports multiple, preemptible threads. A 16 bit timer generates an interrupt once every 1ms or so - the interrupt handler sets motor speed and direction, checks sensor values, looks at whether the time consumed by the currently executing thread has exceeded the set quantum, and if so, stops that thread and chooses a new one for execution. Knowing this much of architectural detail is sufficient to becoming productive with LegOS. Let's look at some code.

Explanations and further exercises will be done in the class. I have to explain a lot of stuff including the setting up of cross development environments, configuration of GCC etc.

4 Code Examples

4.1 Hello, World

```
#include <conio.h>
#include <unistd.h>

int main()
{
  /* Space on the LCD display
   * to print only 'hello' (ie,
   * 5 char's.
   */
  lcd_clear();
  cputs("Hello");
  while(1);
  return 0;
}
```

The classical Hello, World. The message is visible on the LCD screen of the Lego Brick.

4.2 Printing an integer

```
#include <conio.h>
#include <unistd.h>

int main()
{
   /*  Print an int  */
   lcd_clear();
   lcd_int(0xff);
   while(1);
}
```

Displays 255 on the LCD screen.

4.3 Controlling Motors

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>

int main()
{
   motor_c_speed(40);
   motor_c_dir(fwd);
   sleep (10);
   motor_c_speed(off);
   return 0;
}
```

The motors can be connected to any of the three input ports A, B or C. The speed can vary from 0 to

255. Only when the direction is set does the motor start turning. The direction can be one of the enumerated values fwd, rev, off or brake.

4.4 Touch Sensor

The kit comes with two touch sensors. Here is a small program which monitors the state of the touch sensor connected to sensor input 1.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>

/* When depressed, it is 1, else
 * it is zero
 */
int main()
{
   while(1) {
      lcd_clear();
      lcd_int(TOUCH_1);
      /* 500 milli sec sleep */
      msleep(500);
   }
}
```

Here is another program:

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>

#define MIN_ENV_SPEED 40

/* Turn off motors when either
 * touch sensor is touched
 */

int main()
{
   motor_a_speed(MIN_ENV_SPEED);
   motor_c_speed(MIN_ENV_SPEED);
   motor_a_dir(fwd);
   motor_c_dir(fwd);
   while(1) {
      if (TOUCH_1 || TOUCH_2) {
        motor_a_speed(off);
        motor_c_speed(off);
        break;
      }
   }
   return 0;
}
```

Instead of turning off the motors, we can think of reversing the direction.

4.5 Light Sensor

The unit comes equipped with one light sensor, which operates in either active or passive mode. In active mode, the light sensor operates under the glow of a small LED - it is ideal to find out reflectivity changes of near objects. In passive mode, the sensor operates under environmental illumination.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>

#define MIN_ENV_SPEED 40

int main()
{
   /* My desk shows a reading of
    * around 87. The dark side of
    * a floppy around 40 and the
    * metallic part of a floppy
    * around 124.
    */
   /* Get into active mode */
   ds_active(&SENSOR_2);
   while(1) {
     msleep(750);
     lcd_int(LIGHT_2);
   }
}
```

4.6 Wake up functions

It is possible for a thread to get suspended on a wakeup function - the idea is simple - it seems that the scheduler calls the wakeup function upon every timer click - if the function returns true, the thread is woken up.

The program below counts till 9, then stops till a touch sensor (on i/p 1 or i/p 3) is depressed, and after the sensor press, keeps on counting.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>

#define PRI 10

/* Demonstration of
 * wait on an event. Source
 * in tm.c. Fairly simple. Process
 * is suspended till the wakeup function
 * is true */


wakeup_t touch_wakeup(wakeup_t data)
{
   return TOUCH_1 || TOUCH_3;
}

int main()
{
   int i;
   for(i = 0; i < 10; i++) {
      lcd_int(i);
      msleep(500);
   }
   wait_event(&touch_wakeup, 0);
   for(; i < 100; i++) {
      lcd_int(i);
      msleep(500);
   }
   return 0;
}
```

4.7 Basic Threading

A new thread is started by the execi call, which is given the address of the function to be executed as the new thread, an argument count and and argument vector, priority (maximum 20) and a stack size, which is set to a default value.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>

#define PRI 10

/* Main and fun1 execute in
 * parallel
 */
int c = 0;
int fun1(int argc, char **argv)
{
   while(1) {
     cputs("Fun1");
     msleep(250);
   }
   return 0;
}


int main()
{
   execi(&fun1, 0, 0, PRI, DEFAULT_STACK_SIZE);
   while(1) {
      cputs("Main");
      msleep(450);
   }
   return 0;
}
```

Does the termination of the main thread result in the stopping of the ‘child’ thread? No, that’s what I gather from the output of the following program. It keeps on counting.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
```

```
#include <dsensor.h>

#define PRI 10

/* What happens when main dies?
 * Child thread does not stop, it
 * keeps on running. */

int c = 0;

int fun1(int argc, char **argv)
{
   while(1) {
     lcd_int(c++);
     msleep(700);
   }
   return 0;
}


int main()
{
    execi(&fun1, 0, 0, PRI,
    DEFAULT_STACK_SIZE);
    sleep(5);
    return 0;
}
```

The execi function returns a thread id, which can be used by the 'kill' function.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>

#define PRI 10

/* Demonstration of
 * kill */

int c = 0;

int fun1(int argc, char **argv)
{
   while(1) {
     lcd_int(c++);
     msleep(700);
   }
   return 0;
}

int main()
{
   pid_t p;
   p = execi(&fun1, 0, 0, PRI,
   DEFAULT_STACK_SIZE);
   sleep(5);
   kill(p);
   return 0;
}
```

Once again, I have to cover a lot of stuff in the class - including the meaning of concurrency, issues of synchronization etc. I expect things to become a bit confusing.

4.8 Playing Sound

Playing musical notes is simple.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>
#include <dsound.h>


note_t music[] = { {PITCH_D4, 5},
                 {PITCH_END, 0} };

int main()
{
   cputs("start");
   dsound_play(music);
   wait_event(dsound_finished, 0);
   cputs("end");
   return 0;
}
```

We define an array of structures, each structure having two fields, a pitch and duration (seconds). The pitch is defined in standard musical notation (and is internally converted to real frequency). The array should end with the structure {PITCH_END, 0}. All the pitches in the array are played, till {PITCH_END, 0}. The standard function dsound_finished returns true only when the hardware finishes playing the tunes.

5 The counter robot

This is my first non trivial LegOS program - I have built a simple 4 wheeled vehicle driven by two motors. The vehicle moves in a straight line - it has a light sensor attached to sensor input 2. The light sensor points down and counts some brown paper strips (or say black lines) on the floor by measuring change in reflectivity. When the robot hits an obstacle, it turns back and reaches the position from where it started by counting the strips in reverse during the return journey. Each time a strip is encountered, a strip count is displayed on the LCD and a 1 second musical note is played.

```
#include <conio.h>
#include <unistd.h>
#include <dmotor.h>
#include <dsensor.h>
#include <dsound.h>


#define MIN_ENV_SPEED 40
#define PRI 15

/* Our robot should move in
 * a straight line. If the light
 * sensor detects a piece of brown
 * paper, the bot should beep and
 * display a count on the panel.
 */

note_t music[] = { {PITCH_D4, 1},
                 {PITCH_END, 0} };

int strip_count = 1;
int dir = fwd;

wakeup_t strip_found(wakeup_t dat)
{
   return (LIGHT_2 < 82);
}

wakeup_t strip_not_found(wakeup_t dat)
{
   return (LIGHT_2 > 82);
}


wakeup_t touch_pressed(wakeup_t dat)
{
   return TOUCH_1 || TOUCH_3;
}

wakeup_t touch_released(wakeup_t dat)
{
   return (!TOUCH_1) && (!TOUCH_3);
}

int bumper_watch(int argc, char **argv)
{
   wait_event(&touch_pressed, 0);
   motor_c_dir(rev);
   motor_a_dir(rev);
   dir = rev;
   strip_count--;
   return 0;
}

int main()
{
   motor_a_speed(MIN_ENV_SPEED);
   motor_c_speed(MIN_ENV_SPEED);

   motor_a_dir(fwd);
   motor_c_dir(fwd);

   ds_active(&SENSOR_2);
   lcd_clear();

   execi(&bumper_watch, 0, 0, PRI,
   DEFAULT_STACK_SIZE);

   while(strip_count) {
      wait_event(&strip_found, 0);
      wait_event(&strip_not_found, 0);
      lcd_int(strip_count);
      dsound_play(music);
      if(dir == fwd) strip_count++;
      else strip_count--;
   }
   motor_a_speed(off);
   motor_c_speed(off);
   return 0;
}
```

6 Conclusion

Wait for the next part for Lego Network Programming and action with more sophisticated Lego Robots.