LINUXBOT LIBRARY HOWTO

Version 1.0

Alan Winfield, mailto:Alan.Winfield@uwe.ac.uk
Personal home page: http://www.ias.uwe.ac.uk/~a-winfie/

Intelligent Autonomous Systems Laboratory
University of the West of England, Bristol
Coldharbour Lane
BRISTOL

1. Introduction

This howto is for Linuxbot users (i.e. programmers) who wish to make use of the Linuxbot library functions in the latest version of robot.c (and its companion robot.h). These library functions provide access to the robot's sensors and actuators. To be more precise, functions are provided for:

- the motors powering the drive wheels;
- the robot's infrared proximity sensors;
- the motherboard Light Emitting Diodes (LEDs);
- the motherboard status switches;
- the robot's A/D converters (including the battery level monitor), and
- initialisation of i/o hardware, and library code data structures.

In fact 90% of the code in robot.c is actually concerned with the first group of functions, for motor control. The Linuxbot makes use of the powerful HP HCTL1100 motion control devices that, with the robot's optical shaft encoders, provide a number of sophisticated closed-loop control modes. The advantage of these devices is that they offload the job of low-level closed-loop motor control from the main processor. Much of the code in robot.c is concerned with initialising and configuring the motion controller devices for the various closed- (and open-) loop motor control modes. These motor control functions completely remove the need for the Linuxbot programmer to have to become familiar with the low-level programming details of the motion control devices.

Another aim of the Linuxbot library is to allow for common source code across the hardware variants of the robot, such that setting a simple compile-time #define is all that is required to re-compile code for different (and future) generations of Linuxbot.

2. Prerequisites and essentials

DO NOT SKIP THIS SECTION

2.1 Knowledge prerequisites

This howto assumes that you have:

a) A basic working knowledge and are comfortable with command-line Linux[1]. DO NOT be tempted to learn command-line Linux on the Linuxbot. Use a PC instead: i.e. something more forgiving of Linux newbies.
b) Expertise in C programming, and familiarity with the Linux C programming tools: gcc, make etc.
c) A robot together with the means to "telnet" log-in via a wireless LAN. This might be via a laboratory wireless LAN, or with a simple point-to-point (ad-hoc) LAN from a laptop to the robot. The remote workstation, or laptop, can be either MS-Windows or Linux based, since both have telnet and ftp clients. You must be familiar with using telnet and ftp. To "telnet" log-in you will need to know the IP address for your robot.
d) You must also be familiar with the procedures for
   - checking, and if necessary changing or re-charging, the robot's battery;

[1] Note that the Linuxbot does not run X-windows, so prior familiarity with X in Linux will not help.

- powering-up and logging-in remotely, via the wireless LAN and,
- before the battery becomes too low, safely shutting down Linux and powering down the robot.

If you feel uncomfortable with any of these aspects then you must seek help before attempting to use or program a Linuxbot.

2.2 Identify your robot hardware

Before doing anything you first need to identify your robot hardware. Currently there are two generations of robot hardware, both of which can be fitted with the Linux o/s. However the first generation of robot hardware is known as the RASCAL, whereas the second-generation hardware is known as either LINUXBOT (or synonymously MOOREBOT). These two generations of motherboard hardware have a different i/o port address structure and so code compiled for one will not run on the other. If you are in any doubt over which generation robot you have, look for the screen printing on the motherboard which, with the copyright UWE notice, will either show 'Rascal' or 'Linuxbot'. Another way of spotting the second-generation Linuxbot hardware is the presence of the analogue i/o daughterboard mounted in front of the PC/104 stack.

2.3 Get the most recent version of the library code

The most recent version of the Linuxbot library code can be downloaded from the Linuxbot support web page, on the IAS lab's web server. Currently at:
http://www.ias.uwe.ac.uk/~a-winfie/linuxbot/

This will be a zipped file that identifies the version, i.e. robot1_5.zip. You will need to ftp this into your home directory on the robot you are using, i.e. /home/username/. Note that by convention all of the robots are set up with a home directory /home/robot/ (which has the password "robot"), and the robot library code is placed into that directory. If /home/robot/ contains the most recent version of the library then you just need to copy this into your own home directory. Please do not be tempted to compile your own test programs in /home/robot/ as this is reserved as a 'clean' copy of the library and example programs to which all users have access. Only you will have access to your own home directory.

Unzip the library code into your home directory as follows:

```
/home/username$: unzip lib1_5.zip
```

to unpack files robot.c, robot.h and the various example test programs. There will also be a readme file with general notes about recent updates.

2.4 Compiling and running robot code

For simple programs you might just #include "robot.c" into your own robot program, in which case you compile as follows:

```
/home/username$ gcc simple.c –o simple –O
```

this compiles program “simple.c” to executable code “simple”, with compiler optimisation switched off (-O), as advised by the Linux io port-programming mini-howto.

Of course for more sophisticated applications you will prefer to compile “robot.c” separately, in which case you just #include “robot.h” in your own program file(s). You can then use “make” to manage the compilation and linkage process (but this isn’t a tutorial on gcc or make).

If you now try and run this code by, for instance, typing “simple” you will find that the program will in all likelihood fail, either with a nice error message (if you used the recommended code for requesting port access, described below in section 3). Or (if you didn’t) with a rather less elegant code dump. The reason for this is straightforward: in Linux you need to have super-user privileges to be able to access i/o ports. (Because Linux is a multi-user o/s, then having multiple users all directly accessing the same i/o ports would clearly be a disaster.)

Thus, before you can try out your robot program, you need to become the super-user, using the command “su”. By convention, the Linuxbots are installed with a null root password (yes I know this sounds crazy, but these are battery-powered robots). Thus, just type:

```
/home/username$ su
/home/username#
```

whereupon the prompt character changes from $ to #, to show that you are now root. You may now run your program as follows:

```
/home/username# simple
```

Or, as a background task:

```
/home/username# simple &
```

Once you have tested your program by, presumably, observing the behaviour of the robot, you can kill the program with Ctrl-C (if it’s a foreground task) or by using the command “kill”, if it’s a background task. To use kill you will need the process ID (PID) no, which you can get by using the command “ps”. Its then a good idea to exit super-user mode by typing Ctrl-D (or logout), to go back to editing or compilation.

Thus, to reiterate. It is good practice to always log-in under your own username, then in your home directory undertake any editing, program maintenance and compilation. Just become super-user to test run code, but exit super-user mode to return to editing and compilation. Do not be tempted to login and do everything as root. If you do, you are much more likely to damage o/s files, in which case you’ll have some system administration to do to repair the damage. Worse still, you may damage other user’s work.

3. Initialisation and Port Access functions

3.1 accessPorts()

```
/* Request Linux to allow access to the I/O ports.  Return non-zero
```

```
   value PORT_ACCESS_OK if access is granted, or 0 is access is
   denied.  Note: Access is only allowed from a process with root
   privileges (ie superuser). */
int accessPorts( void );
```

Any program that needs to access i/o ports directly must first request permission from the o/s; as already mentioned above the program also needs to be run as super-user otherwise Linux will not grant permission. The Linuxbot library encapsulates the code to request port i/o permission in the function **accessPorts()**.

3.2 resetBot()

```
/* Full Reset of the Robot */
void resetBot( robot *Bot )
{
   hardReset();         /* First hard reset, and enable motors */
   initBotData( Bot );  /* then initialise robot data structure */
   softReset( Bot );    /* soft reset, and go into idle mode */
   initFilter( Bot );   /* initialise motion cntrllr filter params */
   openStop( Bot );     /* make sure we're stopped */
   setLEDs( 0 );        /* and finally clear the LEDs */
}
```

The function **resetBot()**, shown in full above, places the robot into an initialised state, with the motors enabled, but stopped, and the motion controllers in idle (i.e. open-loop) control mode. The first function called, **hardReset()**, asserts the physical reset pin on the motion controller devices and also enables the Linuxbot motors. The second function, **initBotData(Bot)**, initialises the robot data structure which is used by (almost) all Linuxbot library functions. It follows therefore that before calling **resetBot()** we must first declare a data structure of **typedef robot**.

The third function call, **softReset(Bot)**, performs a soft reset of the two motion control devices, initialises their registers and places them in ‘idle’ (i.e. open-loop) mode. The motion controller devices contain digital filters, used by the various closed-loop control modes, and the function call, **initFilter(Bot)**, initialises the filter parameters to the values in the **Bot** data structure. These values are set (by the call to **initBotData(Bot)**) to default values which, for most purposes, give good closed-loop control performance. Thus most programmers will never need to modify the motion controller digital-filter parameters.

Programmers who do want to modify the digital filter parameters are first advised to refer to the HCTL1100 motion controller technical data, in order to fully understand the four filter parameters (sample time T, gain K, zeroes A and poles B). To set new values the programmer would simply change the values in data structure **Bot**, then call **initFilter(Bot)**, before issuing any motion control commands. For details of data structure **typdef robot**, and to see the default values for the four filter parameters see the header file robot.h.

The function **resetBot()** finally forces the motors to stop (in open-loop mode), by calling **openStop(Bot)**, then clears the motherboard LEDs by calling **setLEDS(0)**.

3.3 Linuxbot Program Template

Using the functions and data structure described above we have a basic program template, which programmers are advised to use for all Linuxbot applications, as follows:

```
/* Linuxbot program template */

#include <stdlib.h>     /* needed for printf() */

#define LINUXBOT 1      /* or #define RASCAL 1 */
#include "robot.c"      /* or #include "robot.h" */

int main( void )
{
      robot Bot;        /* declare data structure Bot */

      if ( !accessPorts() )         /* request access to i/o ports */
      {
            printf("Port access denied.. root privileges needed!\n");
      }
      else
      {
            resetBot( &Bot );       /* perform general reset */

            /* now command the Linuxbot. . . */

      }
}
```

4. Motion Control Functions

4.1 General Overview

Since the Linuxbot is a differential drive robot, it follows that each drive wheel can be independently controlled. In fact, each drive motor has its own HCTL1100 motion controller. It is important therefore for the Linuxbot programmer to be able to identify the left and right drive wheels, and the 'front' of the robot, since these are – in effect – defined by the motion control library functions.

Figure 1 below orients the Linuxbot.

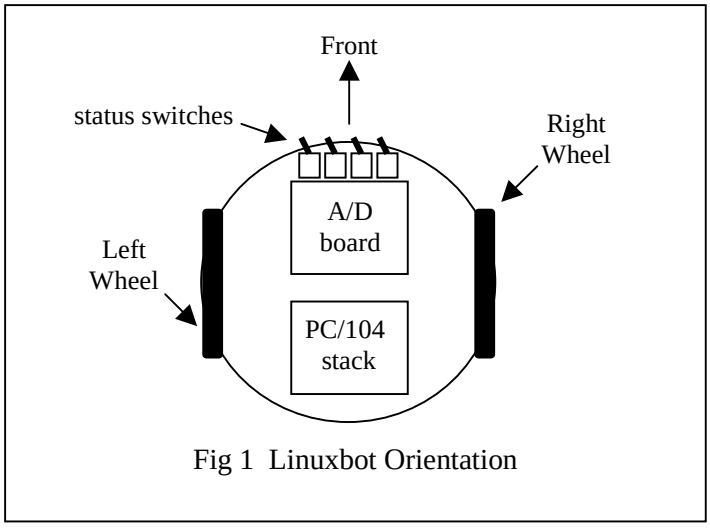

Fig 1 Linuxbot Orientation

Now the HCTL1100 motion controller devices provide one open-loop and four closed-loop control modes. The Linuxbot library provides functions for all five modes of operation. The four closed-loop modes further divide into two groups: velocity control and position control. In velocity control modes the controllers act to maintain a constant velocity, whereas in position control modes the controllers aim to move the drive wheels through a specified distance. The five control modes are summarised in the table below.

Motion control mode	**Library mode selector**	**Mode type**
Open-loop mode	IDLE_MODE	open-loop = idle mode
Proportional Velocity mode	PV_MODE	velocity control
Integral Velocity mode	IV_MODE	velocity control
Position Control mode	PC_MODE	position control
Trapezoidal Profile mode	TP_MODE	position control

At the lowest level the closed-loop control modes operate at the level of shaft-encoder quadrature counts. However, the Linuxbot library provides functions to convert real-world velocities (in metres per second) and positions (in metres) to low-level quadrature values. This howto only describes the top-level real-world value motion control functions, on the basis that the Linuxbot programmer will rarely, if ever, need to access the quadrature value functions directly. Programmers who do wish to use the low-level motion control functions are referred to the robot.c source code.

The top-level motion control functions are described in the following sections.

4.2 setMode()

```
/* Select one of the five operating modes for the motion controllers,
   the default case is Idle (ie open-loop) mode. */
void setMode( robot *Bot, int mode );
```

The function **setMode()** must be called with one of the two mode values listed in the table above, i.e. **IDLE_MODE**, **PV_MODE**, **IV_MODE**, **PC_MODE** or **TP_MODE**. If any other value is provided for the second (mode) parameter, then the function will default to **IDLE_MODE**.

Note that if open-loop motion control only is required then **setMode()** is unnecessary, since the general reset and initialisation function **resetBot()** leave the robot in **IDLE_MODE**.

4.3 velMove()

```
/* Execute a move at given real number velocities, in m/s, for the
   left and right wheels respectively.

   This function uses either one of the two closed-loop velocity
   modes: Proportional velocity mode or Integral velocity mode, or
   if we are in idle mode, then use (the highly inaccurate) open-loop
   PWM control move.

   If the robot is in any other mode, then do not attempt a move, and
   return with the non-zero WRONG_MODE_ERROR.  Otherwise return 0
```

```
   (success).
*/
int velMove( robot *Bot, float LeftV, float RightV );
```

The **velMove()** function is used to command the robot to move, in either open-loop mode (**IDLE_MOVE**), closed-loop proportional velocity mode (**PV_MODE**) or closed-loop integral velocity mode (**IV_MODE**). The function **setMode()** must have been called first to set the motion control mode (but **setMode()** does not have to be called before every call to **velMove()**, only when a change of motion control mode is required).

All three velocity modes need the target left and right drive wheel velocities to be supplied as the 2nd and 3rd parameters in the function call, respectively. These are floating point values in metres/second. Positive values drive the wheels in the forward direction (refer to figure 1), so supplying two equal positive value drives the robot forward in an approximately straight line[2]. Two equal negative values drive the robot in reverse.

Thus, to drive the robot forward at 1m/s, in integral velocity mode:

```
setMode( &Bot, IV_MODE );
velMove( &Bot, 1.0, 1.0 );
```

Or, to turn clockwise on the spot, at 2cm/s, again in integral velocity mode:

```
setMode( &Bot, IV_MODE );
velMove( &Bot, 0.02, -0.02 );
```

It is important for the programmer to understand that after the **velMove()** function has been executed the robot will continue to move at the given velocity forever, or until another **velMove()** (or **resetBot()**) function is executed with different velocity values. Thus, if the robot is moving when the control program is killed, the robot will in all likelihood continue to move until it collides with an arena wall. It is therefore a good idea to have a simple 'allstop' program to hand, ready to run from the Linux command line, ready for this eventuality.

4.3.1 Open-loop velocity mode (IDLE_MODE)

In open-loop velocity mode the real value velocities supplied to velMove() are simply scaled and supplied directly to the PWM registers for the left and right motion controllers. Thus, the robot will move only at very rough approximations of the requested velocities, depending on factors such as the roughness of the floor, or any incline of the surface over which the robot is moving. In fact, in open-loop mode at very low values of command velocity the robot may simply not move at all – because there is insufficient torque from the motors. It is also the case that differences between the left and right motor-gearbox and wheel assemblies will be exaggerated in open-loop mode, so that a straight line move will be much less straight than in the closed-loop modes.

[2] In open-loop mode this will not be a straight line at all, but the two closed-loop control modes will achieve better straight-line performance. Remember that the closed-loop motion control acts only on each motor-gearbox and shaft-encoder assembly, so slight differences in wheel size between left and right wheels, or wheel alignment, or gearbox ratios, etc, will not be corrected.

4.3.2 Closed-loop proportional velocity mode (PV_MODE)

This is a simple proportion velocity mode in which the left and right motion controllers will attempt to achieve the required velocity wheel velocities as rapidly as possible. This can result in very jerky motion, from stopped, and while this may not be a problem in rotational moves, it can cause unstable motion (i.e. wheelies) in a straight line. For this reason, for most velocity moves the integral velocity mode is strongly recommended.

4.3.3 Closed-loop integral velocity mode (IV_MODE)

In the integral velocity mode the motion controller devices ramp up to the desired wheel velocities, at a given acceleration, giving rise to much smoother transitions between different velocities. The acceleration value is provided by the **robot** data structure and is initialised, by **resetBot()**, to the default value in robot.h: 0.8m/s/s.

If a different value of acceleration is required, just modify the **robot** data structure member **mpss** before the call to **velMove()**, as follows:

```
setMode( &Bot, IV_MODE );
Bot.mpss = 0.5;              /* accelerate at 0.5m/s/s.. */
velMove( &Bot, 1.0, 1.0 );   /* to 1.0m/s */
```

4.4 posMove()

```
/* Execute a closed-loop move, by real-number distances LeftD and
   RightD metres, for the left and right wheels respectively.

   This function uses either position control mode, or trapezoidal
   profile mode and assumes that one of these modes has already been
   selected.  If neither of these modes has been selected, then do
   not attempt to move, and return with the non-zero value
   WRONG_MODE_ERROR.  Otherwise return 0 (success).
*/
int posMove( robot *Bot, float LeftM, float RightM );
```

The **posMove()** function commands the robot to move each wheel a given distance, in metres, in either position control (**PC_MODE**) or trapezoidal profile (**TP_MODE**) modes. The function **setMode()** must have been called first to set the motion control mode (but **setMode()** does not have to be called before every call to **posMove()**, only when a change of motion control mode is required).

Unlike the velocity move, a call to **posMove()** should cause the robot to move the required distance and then stop. In fact **posMove()** will return immediately, but the robot will stop (under control of the motion controllers) sometime later. It is therefore the programmer's responsibility to ensure that no further motion control commands are issued before a position move has been completed.

Positive values will move the wheels forward (as defined in figure 1), so to move the robot forward 1m in trapezoidal profile mode, for instance, requires the following code:

```
setMode( &Bot, TP_MODE );
posMove( &Bot, 1.0, 1.0 );
```

Or, to rotate the robot 2cm anticlockwise, in position control mode, requires:

```
setMode( &Bot, PC_MODE );
posMove( &Bot, -0.02, 0.02 );
```

4.4.1 Closed-loop position control mode (PC_MODE)

In position control mode the motion controllers attempt to move the required distance as fast as possible, giving rise to very hard accelerations and decelerations. While this may be acceptable for small rotational moves (for example) its use for long straight-line moves is deprecated. For such moves the trapezoidal profile move is strongly recommended.

4.4.2 Closed-loop trapezoidal profile mode (TP_MODE)

In trapezoidal profile mode the motion controllers attempt to move the required distance using given values for acceleration, deceleration and maximum velocity. The velocity during the move thus follows a trapezoidal profile, as shown in figure 2 below.

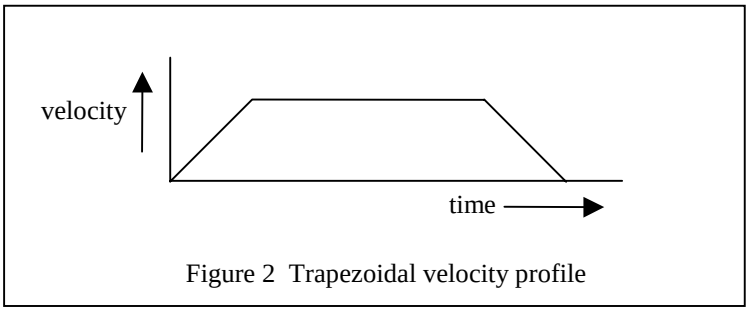

Figure 2 Trapezoidal velocity profile

The two values for the maximum velocity, and the acceleration/deceleration, are provided by the **robot** data structure and are initialised, by **resetBot()**, to default values in file robot.h. The default maximum velocity is 0.5m/s and the default acceleration and deceleration is 0.8m/s/s.

If different values for the maximum velocity, acceleration and deceleration are required, just modify the **robot** data structure members **maxmps** and **mpss** before the call to **posMove()**, as follows:

```
setMode( &Bot, TP_MODE );
Bot.maxmps = 0.2;             /* move at 0.2 m/s and */
Bot.mpss = 0.5;               /* accelerate at 0.5m/s/s */
posMove( &Bot, 1.0, 1.0 );    /* move 1m */
```

Since it is important that a trapezoidal profile move completes before another one is initiated, a special function is provided to wait until a trapezoidal profile move is complete.

4.5 waitforTPMove()

```
/* Wait for a trapezoidal move to complete.  This function MUST be
   called before the 2nd or subsequent trapezoidal profile moves. */
void waitforTPMove( robot *Bot );
```

When called after a trapezoidal profile move, this function simply loops, checking for completion of the trapezoidal moves (on both wheels), and not returning until both moves have been completed.

4.6 rotate()

```
/* Execute a Rotation on-the-spot, by Rot degrees.  Rot MUST be
   within the range -360.0 .. +360.0.  This function assumes we
   are already in one of the two closed loop position control modes.

   Return 0 on success, or WRONG_MODE_ERROR if we're in the wrong
   mode.
*/
int rotate( robot *Bot, float Rot );
```

The function **rotate()** is really just a wrapper function for **posMove()**, and just converts from the required rotation, in degrees, to position values taking account of the physical dimensions of the robot. Positive values for **Rot** produce a clockwise rotation, negative values an anti-clockwise rotation.

4.7 Motion Control Limit Values - IMPORTANT

As a Linuxbot programmer you should be aware that the software might appear to allow unlimited acceleration or deceleration, or very high velocity values. However, the physical limitations of the motor/gearboxes, motor torques, the mass of the robot and other real-world issues clearly limit the maximum values for motion control parameters.

Although the Linuxbot library does provide a degree of hard limiting at the lowest level by virtue of fixed register sizes, maximum register values typically far exceed the capabilities of the physical hardware. You should therefore, in your own robot applications code, check that values supplied to library motion control functions remain within reasonable bounds and cannot runaway.

The following table suggests advisory maximum values for velocities, accelerations and single position moves.

Motion control parameter	**Maxima**	**Control Modes**
Maximum open-loop velocities	± 1.0m/s	IDLE_MODE
Maximum closed-loop velocities	± 1.5m/s	PV_MODE, IV_MODE
Maximum integral-velocity or trapezoidal-profile acceleration	± 1.0m/s/s	IV_MODE, TP_MODE
Maximum trapezoidal-profile mode velocity	± 1.5m/s	TP_MODE
Maximum single position move	± 10m	PC_MODE, TP_MODE

5. Linuxbot Sensor and I/O Library Functions

5.1 getSensors()

```
/* Get the value of the robot's bump sensors */
int getSensors( void );
```

On the Linuxbot[3] there are four digital inputs for infra-red (or other) sensors, and these inputs are mapped to the bottom four bits, (bits 0..3), of the integer value returned by **getSensors()**. The programmer will need to determine which proximity sensors are connected to which input bits (by trail and error, for instance). See the example program avoid.c for code which makes use of **getSensors().**

5.2 setLEDs()

```
/* Set a value to the robot's LEDs,
   for the RASCAL there are 8 LEDS, so this can be a value 0..0xff,
   for the LinuxBot/MooreBot there are just 5 LEDS, so values 0..0x1f
*/
void setLEDs( int value );
```

The Linuxbot has 5 motherboard mounted LEDs, which may be set by the bottom five bits, (bits 0..4), of the parameter **value**.

5.3 getSwitches()

```
/* Get the value of the robot's status switches */
int getSwitches( void );
```

The Linuxbot has four switches mounted at the front of the motherboard (refer to fig 1). These switches may be read by **getSwitches()** and the values will be returned in the bottom four bits, (bits 0..3), of the integer value returned.

5.4 getAnalog()

```
/* Get analog (A/D) inputs...
   For the LinuxBot/MooreBot there are 8 12-bit A/D channels (0..7).
   For the RASCAL there are 4 8-bit A/D channels (0..3).
*/
float getAnalog( int channel );
```

The Linuxbot has 8 12-bit A/D channels, of which channel 0 may be configured – via a jumper on the daughterboard - to read the battery voltage. **getAnalog()** performs an A/D conversion for the given channel, and returns the real-number result in the range 0 to 4.096V.

[3] But not the RASCAL