UNIVERSITY OF CAMBRIDGE
DEPARTMENT OF ENGINEERING

1B Integrated Design Project

IDP Technical Databook

TEAM ...... - ....

**PLEASE RETURN
AT THE END OF
THE PROJECT**

P.J.G. Long
P.R. Palmer

Issue date : 15/09/16

.

Contents

SYSTEM DESIGN 7

D.1 Introduction 7

D.2 Design Process 7

D.3 Overview of Tasks 7

D.4 Characterisation Tests 9

Overall System 9

DD.1 Sensor Position and Control 9

MECHANICAL DESIGN 11

M.1 Introduction 11

M.2 Design Process and Overview of Tasks 11

M.2.1 Vehicle Structure 11
M.2.2 Motors and Pneumatic Actuators 11
M.2.3 3D Additive Manufacture 12

M.3 Robot Manufacturing 12

M.3.1 Workshop Safety 12

Mechanical 13

MD.1 Manufacturing Drawings 13

MD.2 Assembly Drawings 13

MD.3 Electro-mechanical system 14

MD.3.1 Large D .C .Motor specifications (Maxon) 14
MD.3.2 Small D .C .Motor specifications (Mclennan) 14
MD.3.3 Motor Control 15
MD.3.4 Dimensions of motors 15

MD.4 Pneumatic System 16

MD.4.1 Actuators 16
MD.4.1.1 Dimensions of pneumatic actuators 16
MD.4.1.2 Pneumatic actuator specifications 16
MD.4.2 Directional control valves 17
MD.4.2.1 Pneumatic valve assembly connections 18

ELECTRICAL DESIGN 19

E.1 Introduction 19

E.1.1 General . . . 19
E.1.2 System Overview . . . 19

E.2 Design Process 20

E.3 Design Tasks 20
E.3.1 Defining Circuit Function . . . 20
E.3.2 Circuit Embodiment Design . . . 20
E.3.3 Design Documentation . . . 21
E.3.3.1 Circuit Diagrams . . . 21
E.3.3.2 Parts Lists and Connection Diagrams . . . 22
E.3.4 Printed circuit boards (PCBs) . . . 22
E.3.5 Circuit Construction . . . 22
E.3.6 Circuit testing . . . 23
E.3.7 System Integration and Testing . . . 23

Electrical 23

ED.1 Microcontroller Module 23
ED.1.1 I^2C Bus . . . 24

ED.2 Electrical Components 25
ED.2.1 Logic Family . . . 25
ED.2.2 Discrete Components . . . 25
ED.2.3 Infra Red Emitter/Detector Assemblies . . . 25
ED.2.4 Detection/Identification Circuits . . . 26
ED.2.5 Microswitches . . . 26
ED.2.6 Actuators . . . 27
ED.2.7 LED Indicators . . . 27

ED.3 Power Distribution 27

ED.4 DC Motor/Gearbox Units 28

ED.5 IDP Pic module 28

ED.6 Printed Circuit Boards 29
ED.6.1 Line Sensor - Prototyping PCB Configuration . . . 29
ED.6.2 Prototyping PCB Configuration . . . 30

SOFTWARE DESIGN 31

S.1 Introduction 31

S.2 Methodology for Producing Software Systems 31

S.3 Introduction to the use of the Software Library 33
S.3.1 A Simple Example Program . . . 33

S.3.2 The Workstation Programming Environment 34
S.3.2.1 Geany 34
S.3.2.2 Other IDEs 35
S.3.3 The Microprocessor Programming Environment 35
S.3.3.1 Cross-compiling, Downloading and Running Programs 35
S.3.3.2 Cross-compiling robot control programs 36

S.4 36

S.5 Tasks to be performed 37
S.5.1 Formulating the Software Requirements 37
S.5.2 Initial Tasks and Experiments 37
S.5.3 Other Tasks 37

Software 38

SD.1 Additional C++ Language Features 38
SD.1.1 Bit Manipulation 38
SD.1.2 Hexadecimal Values 38
SD.1.3 Classes 39
SD.1.3.1 Constructors and Destructors 39
SD.1.3.2 Overloading of Member Function Definitions 39
SD.1.4 Global Data 39
SD.1.5 Multiple Source Files 40
SD.1.6 Conditional Compilation 41

SD.2 Library - Mobile Robot Link 42
SD.2.1 The `robot_link` Datatype 42
SD.2.2 Initialisation 43
SD.2.2.1 Object Creation 43
SD.2.2.2 Link Initialisation 43
SD.2.3 Instruction Input/Output 43
SD.2.3.1 Link Resynchronisation 44
SD.2.4 Error Handling 44
SD.2.4.1 Error Codes 44
SD.2.4.2 Resetting Error Recording 44
SD.2.4.3 Retrieval of Error Codes 44
SD.2.4.4 Printing Error Descriptions 45

SD.3 Mobile Robot Instruction Set 45
SD.3.1 General I^2C Bus Input/Output Instructions 45
SD.3.2 A to D Converter Instructions 45
SD.3.3 Motor Instructions 45
SD.3.4 Status Register Instruction 46
SD.3.5 Emergency Stop Instructions 46
SD.3.6 General Instructions 47

SD.4 Library - Support Routines 47

SD.4.1 Input/Output to Files 47

SD.4.2 The `stopwatch` Datatype 47

SD.4.2.1 Initialisation 47

SD.4.2.2 Starting Timing 47

SD.4.2.3 Reading the Elapsed Time 48

SD.4.2.4 Stopping Timing 48

SD.4.3 Time Delay 48

SD.4.3.1 The `delay` Function 48

SD.5 The Diagnostic LEDs 48

SD.6 Sensor Simulator PCB 48

SD.6.1 Board Layout 48

SD.6.2 Using the Board 48

SD.7 Details of the robot command interface 49

IDP Kitlist 50

K.1 Introduction 50

K.1.1 Mechanical Materials/Components 50

K.1.2 Electrical Components 51

K.1.3 Software Components 52

1B Integrated Design Project
Mobile Robot – Systems Design

D.1 Introduction

This section describes the system design process for the mobile robot. It is the responsibility of the whole team to clarify the task, produce an overall specification and undertake the conceptual design of the robot. Having produced an overall specification and chosen an appropriate concept, the subteams should then be able to specify, design and build subsystems that can be integrated into the final design.

D.2 Design Process

The various activities and resulting outputs of the design process are shown in Figure 1 below. Within a relatively simple multidisciplinary project such as the IDP, it is usual for the whole design team to be responsible for the design process up to and including concept selection and thereafter for each subteam to concentrate on their particular specialisation.

Useful general guidelines for the design process are given in the following paragraphs – refer to the Part IA handout "An Introduction to the Design Process" for more detail.

The initial task in any design process is to derive a *solution-neutral problem statement*, i.e. a statement which identifies the real problem to be solved whilst avoiding any indication as to how it is to be solved. Having decided on the problem statement, it is then necessary to clarify the design task by identifying the true requirements and constraints. The result of this phase is a detailed list of design specifications – a *requirements specification.*

The next phase is *conceptual design*, where concepts with the potential of fulfilling the requirements listed in the design specification must be generated. The overall functional and physical relationships must be considered and combined with preliminary embodiment features. Sketches of the concepts should be produced.

Subsequently, an *evaluation* of the different concepts must be performed. The evaluation is based on the wishes identified in the design specifications. More than simple yes/no answers are required to determine the relative merits of each concept. An evaluation chart, where each criterion is weighted to indicate its relative importance for each concept, should be produced.

Where a design is to be the responsibility of more than one person, and, in particular, when the design task is multidisciplinary, as in the IDP, it is important for each team member and subteam to be clear which aspects of the design are their responsibility and to agree the *interfaces* of their design with the other team members and subteams. Having decided an overall specification and concept as a team and agreed the interfaces of their design with the other subteams, it should then be possible for each subteam to proceed with their particular design process.

Although the model shown in Figure 1 implies a sequential process, in practice design is very rarely so. For example, it is often necessary to amend the product specification – perhaps as a result of changes requested by the market or as a result of additional information gleaned as part of the design activity itself.

D.3 Overview of Tasks

The main design tasks on the IDP are the *vehicle structure* (its motion and its object handling/transportation functions), *the electronics interface* and the *software control program.* Each of

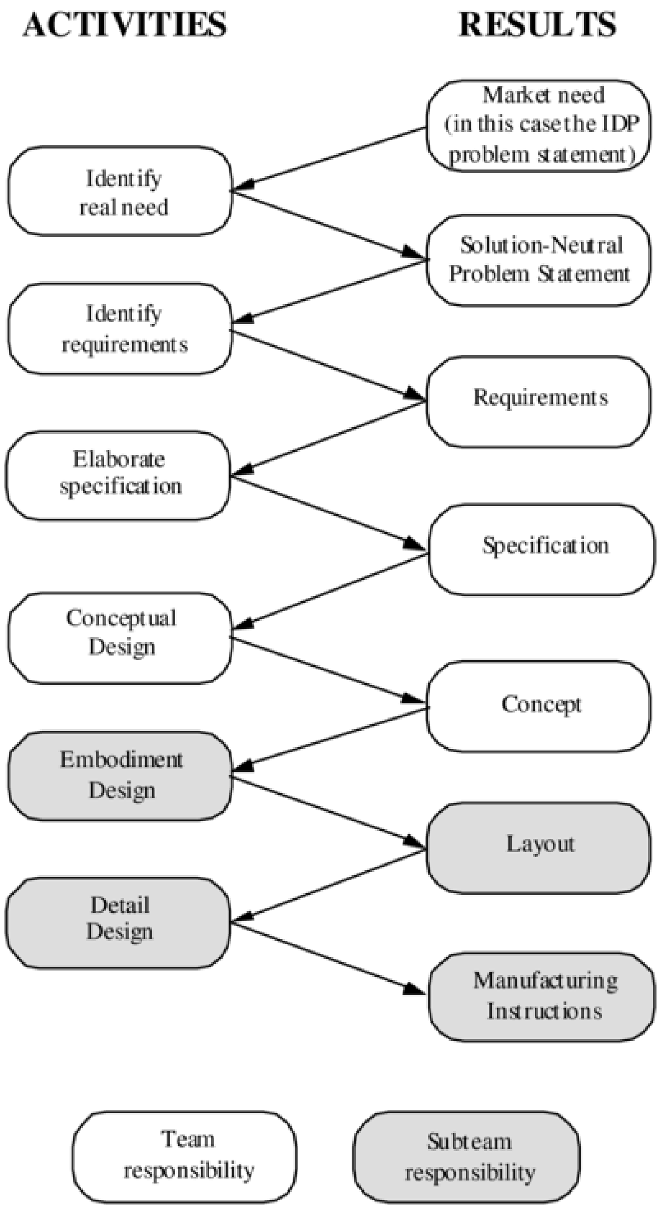


Figure 1: The Design Process

these are subject to constraints which will need to be identified and incorporated into the design. These include general constraints, such as the recyclability of the robot, as well as specific ones, such as the limited materials and components.

In particular, the following issues should be considered in the first stages of the robot design:

- The overall strategy, including the software control strategy
- The choice of wheel base (the layout, number of wheels, front or rear wheel drive)
- The choice of drive system, gear boxes and wheel sizes (the effects of these on traction, vehicle speed, stability etc.)
- The design of the object handling system (the adequacy of lifting forces, its reliability and effectiveness, its interaction with the drive system)
- The electronics interface between the microprocessor and the 'real world'. The use of sensors

(which of the available sensors fulfil the required functions best?)

- The positioning and mounting of sensors (see the IDP website for more information)
- The overall manufacture of the design (the appropriate construction sequence and timetable, anticipated difficulties etc.)
- Energy management of the system
- Risk evaluation. Consider carefully your proposed strategy and design – what are the greatest risks to a successful outcome and what measures can you take to minimise these?

These decisions will have implications for the whole team. Thus, on-going consultation with your team members is essential.

D.4 Characterisation Tests

There are a number of experimental rigs available in the project laboratories. These are mostly specialised test facilities which are described in the relevant subteam documents. In the EIET Lab there are rudimentary robots and a pneumatics test rig which may be useful to the team as a whole in characterising the motors, pneumatics and associated equipment. Ask demonstrators for guidance on using these experimental rigs.

DD.1 Sensor Position and Control

There are no rigid rules to apply for the control of wheeled vehicles, but simplifying models can give valuable insight into the positioning of sensors, and the control parameters for line following. In your project, there are two types of controllers which you can build: (i) simple on / off controller and (ii) pseudo proportional controller .

Various sensor layouts can be used to optimise line-following or junction detection and there will often be trade-offs between the two. For line following, the sensors can provide varying degrees of proportionality from a single sensor providing on/off indication of whether the robot is on the line; through two providing on/off indication of whether the robot is on the line and to which side it has moved off; to three providing a degree of proportionality according to how many are on the line; to arrangements in which the height and positioning of each sensor is arranged to give an analogue representation of how much it is over the line. Investigating sensor layouts should be a key early design activity for the team.

Model of linear controller

A sensor S gives an output voltage proportional to its distance y from a specified path. The sensor is attached to a vehicle steered with angular velocity $\dot{\theta}$.

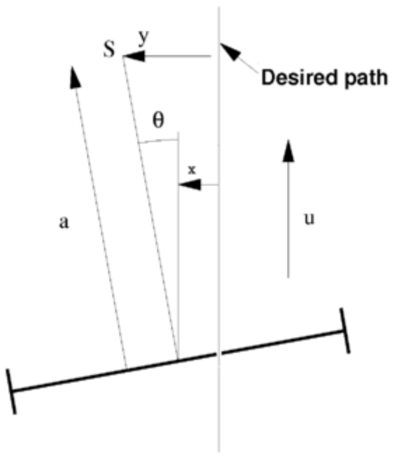
S	Sensor position
u (m/s)	forward velocity
θ (rad)	heading angle
y (m)	distance of sensor S from desired path
a (m)	distance between sensor and driving wheel axle
k (l/ms)	controller gain

The equations of motion for the controller, for a small angle θ, can be easily derived to be

$$\ddot{\theta} + ka\dot{\theta} + ku\theta \quad = \quad 0 \quad \text{for straight line operation}$$

These equations describe damped simple harmonic motion, with damping ration $\zeta = \frac{a}{2}\sqrt{\frac{k}{u}}$ and hunting frequency $\omega_n = \sqrt{ku}$.

For a constant gain k investigate qualitatively (i.e. without estimating a value for k) the effect of the sensor position on the damping ratio and the motion of the vehicle. (Hint: Is it useful to have



damping? If so, which is preferable: *overdamped, critically damped* or *underdamped* motion?) Also, what is the constraint on the sensor position when cornering?

Irrespective of the controller you wish to use, the discussion given below of a simple linear (proportional) controller gives an insight into the appropriate positioning of the sensors and the resulting motion of the vehicle.

1B Integrated Design Project
Mobile Robot – Mechanical Design

M.1 Introduction

This section describes the mechanical design and construction of the mobile robot. The aim of the mechanical subteam is to design and build the robot's chassis and object handling system and, in conjunction with the other subteams, test the robot which will be controlled via on-board electronics and software routines.

The kit of parts provided includes motors, pneumatic actuators and structural materials. Fasteners, lubricants, adhesives and other components, such as springs and gears, are also available.

Useful guidelines for prototype design and manufacturing are given in the handout 'Practical guidelines for mechanical design and manufacturing', copies of which can be found in the DPO and EIET Lab and on the IDP website.

M.2 Design Process and Overview of Tasks

As part of the overall system design undertaken by the whole team, you should have clarified the task, produced an overall specification and selected a design concept. The next phase of the design process is the embodiment design of the mechanical aspects of the robot, where the foundations are laid for detail design through a structured development of the concept. The result of this phase is a detailed layout drawing, showing the preliminary shapes of all the components, their arrangement and, where appropriate, their relative motions.

Finally, in the detail design phase, the precise shape and dimensions of every component have to be specified, and the material selections made or confirmed. You are then in a position to start manufacturing. Manufacturing drawings (with an accompanying parts list) should be produced for every component to be manufactured. Individual drawings must be accepted (and the accompanying assembly drawing at least checked) by a demonstrator before construction of that component can be commenced in the workshops.

The main mechanical design tasks are the vehicle structure, the detailed drive system and its object handling/transportation functions. Each of these are subject to constraints which will need to be identified and incorporated into the design. These include general constraints, such as the recyclability of the robot, as well as specific ones, such as the limited materials and components. Your decisions will have implications for your team members working on the software and electronics design. Thus, on-going consultation with your team members is essential.

M.2.1 Vehicle Structure

The vehicle must conform with the requirements of the problem statement. The vehicle must carry the electronics and the micro-controller unit and, if pneumatic actuators are used, the vehicle must also carry a spool valve assembly (providing the link between the compressed gas line and the actuators). The attachment of the micro-controller unit to the vehicle should be made using the bracket provided. Note that the attachment of the umbilicals to the vehicle must be secure and should be positioned so as to cause least inconvenience to the vehicle when in motion.

M.2.2 Motors and Pneumatic Actuators

The specifications and characteristics of the motors and pneumatic actuators are included on the IDP website. Note that compressed gas can only be used in connection with the spool valves to run the actuators. A test rig is available in the EIET Lab for testing your pneumatic actuator assembly separately.

It is a requirement of the design that all main assemblies provided (including the motors and actuators) be re-usable on completion of the project. No permanent modification of the components is permitted (e.g. gluing of the wheels onto the motor shaft). The motors may not be modified. Wheel-motor clamps are provided in the kit of parts.

You will need to work in conjunction with the software subteam in order to determine experimentally appropriate embodiment details of the drive system.

To design the actuator system for object handling, you may find it useful to refer to the 'Practical guidelines for mechanical design and manufacturing' handout for some examples of linkages, pivots and spring arrangements. This is available in the DPO and the EIET Lab and on the IDP website.

M.2.3 3D Additive Manufacture

The use of the Dyson (PLA) or IDP (ABS) 3D printers are allowed, but there is a limit of a maximum of 40cm^3 or 30g of 3D printed material on any single robot. Before attempting detailed design please speak to a Mechanical demonstrator to confirm design principles and manufacturing techniques.

M.3 Robot Manufacturing

Although the design of the vehicle and object handling mechanism should be considered as a whole, it may be advantageous to build the basic vehicle (chassis and motor/wheel arrangement) before the object handler in order that the software and electronics members of your team may start testing the control of the vehicle.

It is important to note that the parts in the kit will not be replaced unless found to be defective. Cardboard is available for model testing. In addition, parts such as springs and gears are available on request but they are of limited supply.

Structural materials as listed in the mechanical kit list may be obtained from the IDP area in the Dyson centre. Additional items may be obtained at Design Acceptance via a BOM. Note that there are fixed limits as to the quantity of each material available.

Guidelines on manufacturing techniques can be found in the handout entitled "Practical guidelines for mechanical design and manufacturing" (which is available in the DPO and the EIET Lab and on the IDP website).

M.3.1 Workshop Safety

When using the workshop, you must follow important safety procedures:

THE WORKSHOP IS AN EYE PROTECTION AREA. PLEASE WEAR YOUR SAFETY GOGGLES WHEN OPERATING MACHINERY.

Before operating any machine please ensure:

1. You are not wearing loose clothing
2. Long hair is tied back
3. The workpiece is securely clamped
4. If in doubt, check with the workshop staff before commencing work

Note: Compressed gas is potentially dangerous; please use the airline and pneumatic control lines with care.

MD.1 Manufacturing Drawings

Manufacturing drawings should be produced for **every component** which you have to manufacture (i.e. every component which is not a standard item from the kit list). If you simply need to cut, say, some threaded rod to a certain length, it is acceptable just to list this in the parts list on the relevant assembly drawing (e.g. "M4 threaded rod 125 mm long") but a manufacturing drawing must be produced for any component which has to be cut from sheet or angle and drilled or bent or printed. For components to be made from sheet the drawing should be of the component in flat form.

The dimensions necessary to locate all holes to be drilled, the size of holes, the positions of bends etc. must be included on the manufacturing drawing. If reference must be made to another drawing in order to establish a dimension, the manufacturing drawing will not be accepted.

Dimensioning should reflect the underlying design requirement. For instance, in the simple bracket drawing below the distance between the centres of the $\phi 4$ holes is stipulated to be 10 mm reflecting the fact that the sensor to be mounted on the bracket has two holes 10 mm apart which must align with these. The centres of both $\phi 4$ holes could have been shown as being 5 mm from the edges of the bracket, but this dimensioning does not convey the same information about the underlying requirement.

The positions of bend lines must be clearly indicated, but, apart from this essential information, all other dimensioning should be done with respect to physical edges, so, for instance, the positions of holes should not be dimensioned with respect to bend lines.

MD.2 Assembly Drawings

Assembly drawings should be produced to show how components are to be fitted together. They should be include a parts list and be drawn to scale, but, if the individual manufacturing drawings have been done correctly, they should not need to be dimensioned in detail.

For components which you have manufactured, the 'Specification' column of the parts list should be used to reference the relevant manufacturing drawing.

MD.3 Electro-mechanical system

MD.3.1 Large D .C .Motor specifications (Maxon)

(approximations only, since motors vary) :

Nominal voltage	=	12 V	Current drawn by each motor:
Max power	=	4 W	At low speed ∼ 0.05A
Max continuous operation current	=	0.5 A	At med speed ∼ 0.12A
Starting Vmin no load	=	0.15 V	At high speed 0.15A
No load current	=	15 mA	With motors slipping (e.g. hitting wall) ∼ 0.3.A
Terminal resistance of motor windings	=	10Ω	With motor stalled ∼ 1.2A

200:1 Gearbox Specification			100:1 Gearbox Specification		
Max torque: continuous	=	0.6 Nm	Max torque: continuous	=	0.6 Nm (see notes below)
Max torque: peak	=	1.8 Nm	Max torque: peak	=	1.8 Nm
Rated speed	=	20 rpm	Rated speed	=	40 rpm
Reduction	=	200:1	Reduction	=	100:1
Mass	=	0.25 kg	Mass	=	0.25 kg

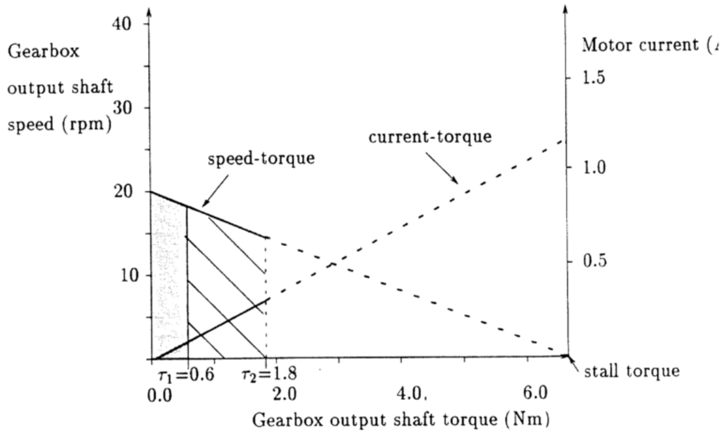


Figure 2: Approximate speed-torque and Current-torque relations for motor with the 200:1 gearbox.

Notes on the curves:

1. Continuous rotation with output shaft torque above τ_1 causes overheating in the gearbox
2. Output shaft torque above τ_2 will cause gear teeth to break;
3. For both the 20 rpm and 40 rpm motor/gearbox arrangements. the maximum output torque is limited by the shear strength of the gearbox components and not the maximum torque or maximum continuous operation current available.
4. What are the corresponding relations for the motor with the 100:1 gearbox?

MD.3.2 Small D .C .Motor specifications (Mclennan)

(approximations only, since motors vary) :

Nominal voltage	=	12 V	
Max power	≈	1.8 W	(At stall)
Rated operating current	=	85 mA	(At full speed)
No load current	=	20 mA	

90:1 Gearbox Specification			43:1 Gearbox Specification		
Rated torque: continuous	=	8 Ncm	Rated torque: continuous	=	3.8 Ncm
Rated speed	=	18 rpm	Rated speed	=	40 rpm
Reduction	=	90:1	Reduction	=	43:1
Mass	=	0.058 kg	Mass	=	0.057 kg

MD.3.3 Motor Control

In order to adequately control the motor speeds, a parameter/speed curve should be obtained experimentally for both gear ratios (to be done in conjunction with the software team see software handout for detail).

You may also be interested to know that:

Maximum pushing force	=	$\frac{\tau_{max}}{r}$	Maximum speed	=	$r\omega_{nolood}$
Traction of flat surface	=	$\frac{1}{2}\mu Mg$	Traction of slope	=	$\frac{1}{2}\mu Mgcos\theta$

where τ_{max} is the maximum torque, ω_{noload} is the no-load motor speed, r is the wheel radius, μ is the coefficient of friction between the surface and the wheel and M is the mass of the vehicle.

Note that (i) the pushing force is multiplied by two for two motors, (ii) you go faster with larger wheels, (iii) the traction calculations assume the centre of mass of the vehicle is in the middle of the two motors, and that the centre of mass is close to the ground. You can do better by raising the centre of mass so that the line of action of its weight passes through the driven wheels, but then there is danger of the vehicle tipping over.

MD.3.4 Dimensions of motors

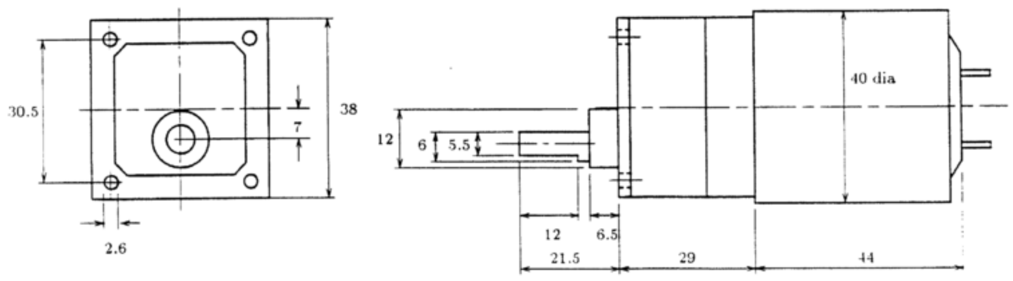


Figure 3: (Maxon) Large 20/40 rpm motor.

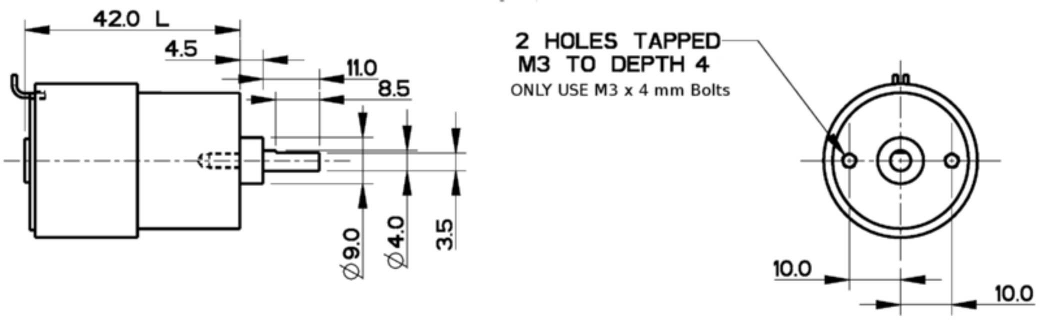


Figure 4: (Mclennan) Small 18/40 rpm motor.

MD.4 Pneumatic System

MD.4.1 Actuators

Actuators convert-fluid energy into mechanical work, and fall into two main categories: single and double acting. A single acting cylinder uses the working fluid to extend the piston/shaft and an internal spring (or the load) for retraction. A double acting cylinder uses the working fluid to extend and return the piston/shaft. In this project the working fluid is compressed air.

Each kit contains two double acting actuators complete with pneumatic connections and flow adjusters. (NB An alternative end fittings are available, type 1 and 2, see below)

MD.4.1.1 Dimensions of pneumatic actuators

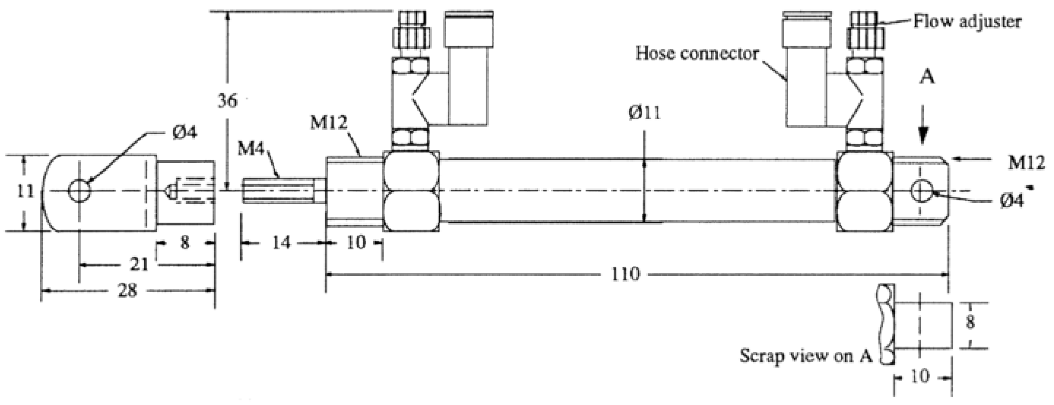


Figure 5: Actuator Type 1

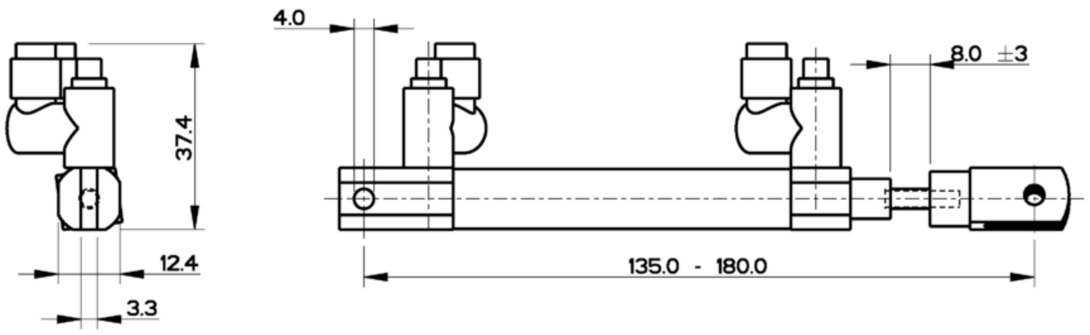


Figure 6: Actuator Type 2

MD.4.1.2 Pneumatic actuator specifications

Actuator type	Double acting
Bore (b)	10mm
Stroke	45mm
Shaft diameter	4mm
Operating pressure	4 bar (60 pounds per square inch)
Actuator and clevis mass	90g

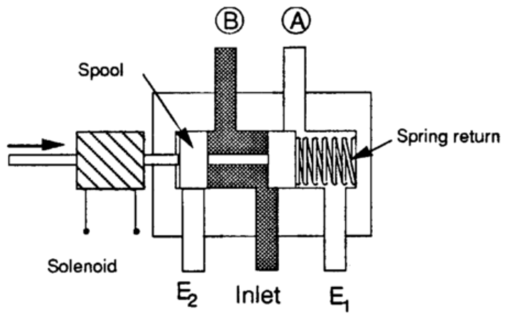
Note that a double acting actuator means that pressure is needed to activate the piston for both the inward and outward stroke.

The force acting on the piston on the outward stroke is proportional to the piston area and the gas pressure ‘P’

$$F = \frac{\pi b^2}{4} P$$

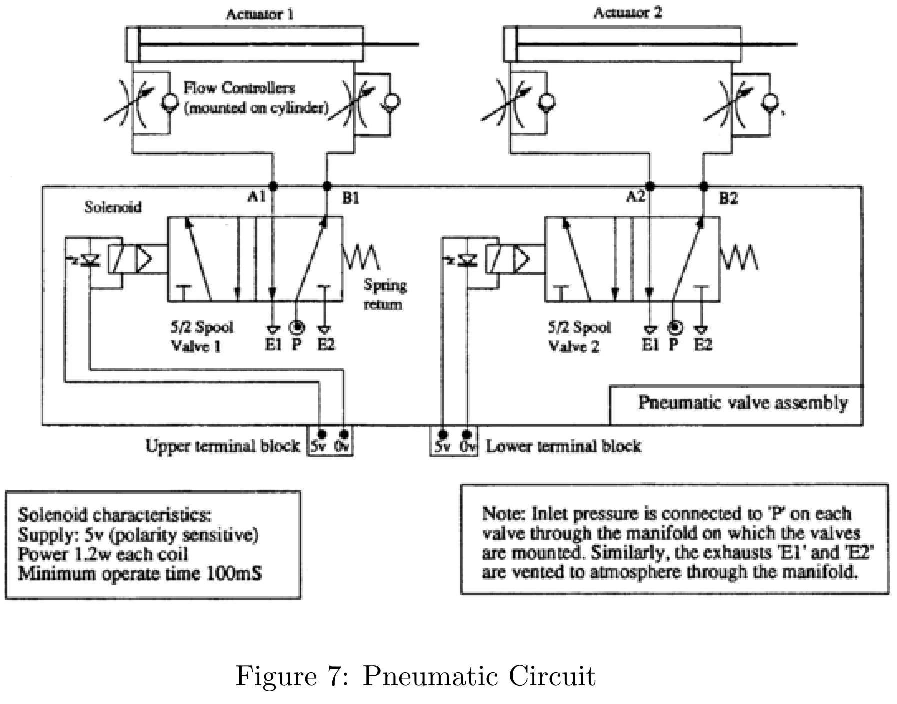
Is the force the same for the inward stroke?

MD.4.2 Directional control valves



Control valves are used to switch the compressed gas to the required actuator port. Types of control valve include sliding and rotary spool valves. The valve used on this project is a five port, two way (5/2) solenoid operated, spring return spool valve (shown schematically below).

The valve is shown with the solenoid unenergised and the spool fully retracted by the spring. In this state the inlet pressure is connected to port B and port A vents to atmosphere via E_1. When power is applied to the solenoid, the spool moves in the Spring return direction of the arrow, connecting port A to the / inlet pressure and port B vents to atmosphere via E_2. E_1 and E_2 are internally connected to the vent to atmosphere. The spool valves are mounted, together with the pneumatic and electrical connectors, in an enclosure measuring 80 x 100 x 50 and weighing 200g approx.

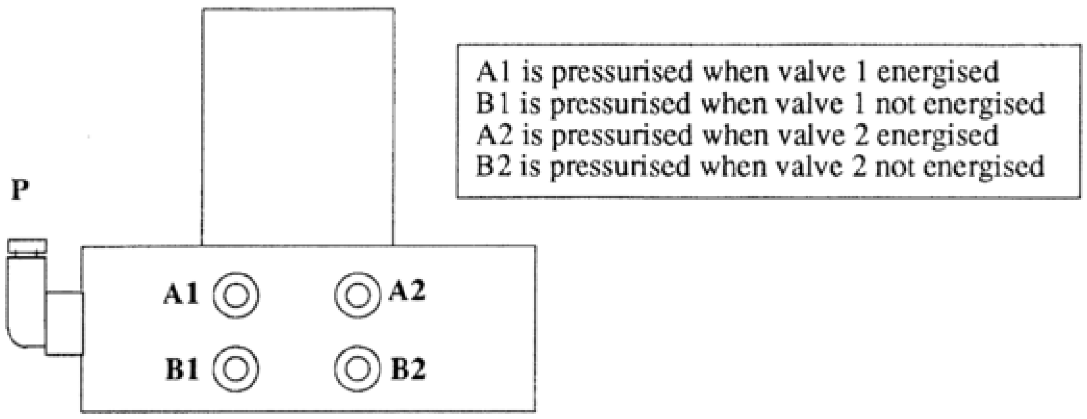


Solenoid characteristics:
Supply: 5v (polarity sensitive)
Power 1.2w each coil
Minimum operate time 100mS

Note: Inlet pressure is connected to 'P' on each valve through the manifold on which the valves are mounted. Similarly, the exhausts 'E1' and 'E2' are vented to atmosphere through the manifold.

Figure 7: Pneumatic Circuit

MD.4.2.1 Pneumatic valve assembly connections



A1 is pressurised when valve 1 energised
B1 is pressurised when valve 1 not energised
A2 is pressurised when valve 2 energised
B2 is pressurised when valve 2 not energised

1B Integrated Design Project
Mobile Robot - Electronics Design

E.1 Introduction

E.1.1 General

This section describes the electronic part of the Mobile Robot Project. The aim of the electronics sub-team is to design, build and test the interface electronics for a mobile robot. The resulting electronic circuits will be the interface between the robot's transducers, which provide information on the 'real world', and the robot's on-board microprocessor.

A kit of parts is supplied, containing two circuit boards on which your circuit is to be built. One is a pre-configured board for a line following circuit with a limited amount of prototyping space, the second is a prototyping board. The use of two boards allows flexibility in the construction of the electronics and testing of the robot. (A further prototype board is available if the space is required and can be justified to a demonstrator). You will also have a range of electronic components available from simple resistors and compacitors through to CMOS logic, operational amplifiers and a programmable logic module. The transducers that can be used include infra-red/magnetic/strain sensors and micro-switches. Electronic interfaces are also required for any actuators.

The sequence of the design process for the interface electronic circuits is outlined in this document. Detail on the prototyping boards, individual assemblies and components and their usage is given on the IDP website. Full data on the ICs and transducers and the actuator circuit is provided in the Electronic Component Data section of the website and may be found on the internet.

The marking of the project will take into account the approach to the design process and the quality of the final circuit. Assessment will be based on the clarity of the circuit specification and diagrams, the reliability and layout of the complete circuit and its ability to perform the task specified. This is an integrated design project, and careful co- operation is required between those dealing with the electronic, software and mechanical aspects of the design.

E.1.2 System Overview

The completed prototyping boards are connected to a microcontroller module (see the IDP website) via a standard communication bus known as I2C (Inter-Integrated-Circuit bus. The module also includes a wireless interface (802.11b) enabling communication between the microcontroller and a workstation or terminal on the Engineering Department's teaching network.

Electrical power comes from a 12v bench power. The supply to the microcontroller and other electronics is regulated to 5V, ±15v in the microcontroller unit. The module and prototyping boards are connected via 9 way D plugs/sockets which carry the I2C bus, +5V, ±15v + unregulated +12v and a ground line; the current in this 5V supply line to the prototyping boards should be design not exceed 0.4A.

When the vehicle is operating in its working area the 12V power line can be supplied via an umbilical, which also supplies the pneumatic supply.

There are power drivers for two(/four) drive motors in the microcontroller. The motor terminals should be linked directly to the connectors on a drop cable supplied with the microcontroller module.

Pneumatic Actuators, which may be used to power the pick and place mechanisms, are powered by the pneumatic supply. The pneumatic valves are controlled using electrical signals from the prototyping board or the prototyping area on the 'line sensing' board.

E.2 Design Process

The interface electronics circuit design process can be split into stages:

- Definition of circuit functions.
- Circuit embodiment design and calculation of resistor and capacitor values.
- Circuit documentation.
- Circuit construction and test.
- System integration.

All the information you should need on the I2C bus and the transducers is provided in the latter sections of this document and on the web. The ICs and components listed in the kit section (K.1) of this document form part of the basic kit for the project.

The operation of the interface electronic circuits can be considered under the following headings:

- Line following using infra-red emitter/detector assemblies: The robot's path in the working area is defined by white lines on the black surface. The infra-red sensors provide the information on the robot's position relative to the line.
- Obstacle and target detection using microswitches or infra-red emitter/detector assemblies: Microswitches can be used to detect contact between the robot and an obstacle or target. The infra-red assemblies can be used to detect the presence of a reflecting obstacle without contact.
- Detection/Identification - for example, route selection by detecting signals from transmitters or load sorting by use of sensors.
- Control of 'pick and place' mechanism.
- Light emitting diode indicators to show sensor or actuator status.

E.3 Design Tasks

E.3.1 Defining Circuit Function

The circuit function must be defined as a team activity. The circuit function follows from the vehicle specification required to carry out the problem specified as the task for your robot.

Figure 8 shows the main functional blocks of a complete interface circuit. The only part of this circuit which is already defined is the link to the I2C bus, which must be through the PFC8574 8-bit input/output port ICs. The 8574 port addresses are defined by links on the prototyping and line sensor boards and must correspond to the addresses used in the software.

The first task of the electronics sub-team is to define the type and number of sensors and actuators that you will need to perform the functions that your robot requires. You can then define the signal processing sub-circuits. Your complete circuit should:

- control the infra-red emitting diodes and ensure reliable detection of the infra-red signals for high and low levels of reflection and with the possibility of cross-talk between adjacent sensors,
- ensure reliable detection of microswitch operation in the presence of contact bounce,
- detect any parameter (eg IR beacon, load material) that defines the task to be performed,
- Ensure the current supplied to the 'pick and place' driver is adequate.

There may be some advantages in combining the states of two or more sensors using logic gates rather than using software. The hardware/software trade-off should be discussed during the initial design stage.

E.3.2 Circuit Embodiment Design

As shown in Fig. 1, the sub-circuits link the sensors and actuators to the 8574s and via the I2C bus to the microcontroller.

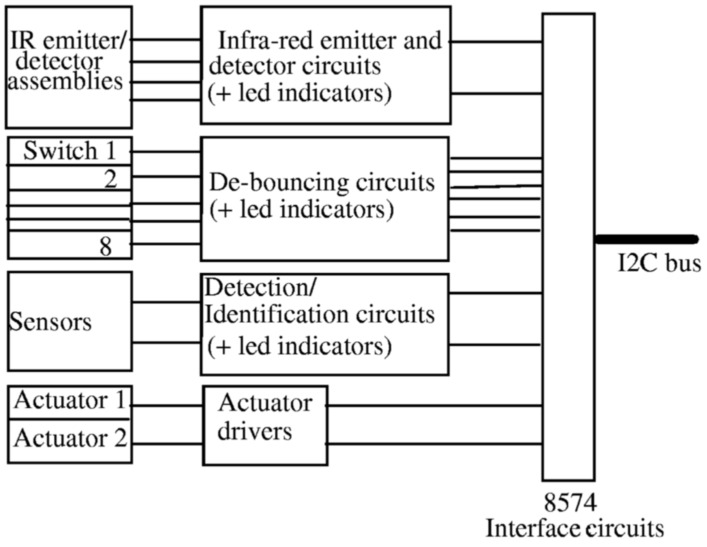

IR emitter/ detector assemblies
Infra-red emitter and detector circuits (+ led indicators)
Switch 1
2
8
De-bouncing circuits (+ led indicators)
Sensors
Detection/ Identification circuits (+ led indicators)
Actuator 1
Actuator 2
Actuator drivers
I2C bus
8574 Interface circuits

Figure 8: Block diagram of sensor interface circuits

Define the operation of each sub-circuit in terms of its function, using logic statements and timing diagrams if necessary. Note carefully whether a given sequential logic component is level- or edge-triggered and which trigger polarity is being used. Use your notebook to record this information and any calculations that you make.

Then devise a circuit which implements the functions. Some guidance on the form the circuit may take is given in the Appendix to this document. You should include LED indicators to show the status of certain sensors or actuators. These indicators can help considerably when you are testing the complete vehicle. Using the information and tables provided in the product Data sheets produce a complete circuit diagram, showing which gates you are using within each IC package. Calculate and record the values required for resistors and capacitors.

Prepare a table showing any significant current requirements (say greater than 10 mA) of individual devices. Estimate the expected 5V supply current required by your board under worst case load.

E.3.3 Design Documentation

E.3.3.1 Circuit Diagrams

Circuit diagrams should be prepared according to the following guidelines:

- Standard symbols should be used.
- All interconnections should be clearly indicated, both within and between circuit blocks, and including power supply connections.
- Interconnections should be shown as an interconnecting line, or by allocating a unique label to a signal and then labelling ”hanging” interconnections, or by a sub-table of connections (eg as may be used to list power supply connections).
- Unused devices within ICs (eg unused gates) should be shown, with inputs defined high or low as necessary.
- Unconnected pins on ICs should be included, and labelled NC (no connection).
- If power supply distribution lines are shown, the most positive should appear towards the top of the page, and the most negative towards the bottom.

Where a single IC has several gates or other functions, ensure that only one set of power lines is shown for that particular IC. Pin numbering is essential and will be of assistance when constructing the connection diagram and circuit. Demonstrators can asdvise on ideas as to how to draw this in a clear manner (there is no standard approach).

E.3.3.2 Parts Lists and Connection Diagrams

A parts list and a connection diagram showing the components and their orientation where appropriate (eg LEDs, ICs, diodes, MOSFETs etc) and the wiring needed in your circuit should be prepared.

Use a consistent scheme for the orientation of the IC carriers. Plan the layout of the IC packages to reduce the length of interconnections as far as possible. The circuit should be easy to check, test and modify. Ensure that your indicating LEDs will be visible when the circuit board is mounted on your vehicle. An A4 copy of the prototyping board is available to help plan the layout.

If your documentation is complete, then a competent technician will be able to build the circuit exactly to the stated requirements. Complete documentation is also required when assistance is being sought.

Your final report (see the logistics document) must contain the overall circuit diagram, component connection diagram and parts list and each must appear on a single sheet of A4.

Ask a demonstrator if you are uncertain about any stage of the design.

E.3.4 Printed circuit boards (PCBs)

See **Week 1 Tasks and Exercises** section of the IDP manual.

E.3.5 Circuit Construction

Design acceptance is needed before starting circuit construction - see section logistics document.

Circuits can be developed on either of the boards supplied, i.e.

1. The line sensor board which is split into two sections a pre-configured area for a standard optical sensor circuit and a small prototyping area, see section ED.6.1.
   [NB Pre-built versions of the board are available for 'hire', prior to manufacture. However, (a) teams must have demonstrated a working LED drive circuit on breadboard, (b) The use will be taken into account during the assessment of the robot quality and (c) an additional 'consultancy' charge will be levied for inclusion in the overall cost sheet.]
2. The prototyping board is also split into two distinct areas, the prototyping area and an area dedicated to the I2C bus interface circuits, see **IDP Technical Databook**.

The prototyping areas are arranged in 2 columns (1 column on line sensing board). Two continuous tracks for the +5V and 0V lines run in the centre of each column. A standard dual-in-line IC can be mounted so that each pin is linked to two connecting holes. Connections between pins and discrete components are to be made with suitable gauge wire and soldered joints. Within the dedicated area a socket for an 8574 IC is required to be mounted on the board; the port address may be selected using jumper connectors.

The I^{2}C bus and power for the board is supplied via a D plug mounted on one end of the boards. (N.B. These connections are continued through tracks on the board to the socket at the other end enabling more than one board to be used.

The infra-red and actuator assemblies will be provided with leads attached; you will be required to wire the appropriate connectors on to the board, together with the via pins and the D plugs/sockets supplied with each board.

Use IC carriers both to avoid damage to the circuits during the soldering and so that the circuits can be re-used.

Wire the circuit in stages, taking each function in turn. Use single-core wire for the board and multi-core for the connections to external components. You may find it helpful to use different wire colours for each function. Then test the function and return to wire the next stage.

E.3.6 Circuit testing

When you have completed a stage of wiring, check with the multi-meter that the power supplies are routed to the correct pins on the IC holders and sensor connectors before inserting the ICs and sensors. Always switch off the power before inserting or removing an IC or a sensor.

Take each functional block in turn and establish that the function is correct. Generate suitable test procedures without using the microcontroller initially, bearing in mind that the 8574 output pins will be either logic 1 or logic 0. Use the oscilloscope to check any timing sequences based on a clock output. If you find your circuit is not working correctly, devise a methodical approach to find the fault. Ensure first that the power and ground are properly connected. Then systematically trace signals through the circuit from their point of origin.

Ensure the design of the infra-red sensor circuits can cope with small variations in the 'black' and 'white' reflected light levels: changes in mounting height, marks on the white tape and scuffs on the black table surface can cause problems if your design is not tolerant. If a pneumatic system is used, confirm that the actuator driver will operate the control valve solenoid and measure the solenoid current.

E.3.7 System Integration and Testing

When you are satisfied that the circuit works to its design specification, try to link it to the microcontroller. The software sub-team should have written test procedures which will allow you to confirm that the required functions are all operating correctly before the complete mechanical assembly is available. N.B. The team is supplied with a sample chassis which can be use to test the system.

The final tests of your circuit must be designed and carried out as part of the functional tests of the complete vehicle assembly.

ED.1 Microcontroller Module

The Balloon 3 microcontroller module is designed around an Intel Xscale PXA270 microcontroller and contains the cpu plus its supporting circuits. The Xscale is mounted on an approximately 100 x 50 mm printed circuit board. It is a ARM based 32-bit microcontroller that has been designed to have a simple instruction set and minimise power.

Attached to the main board is a single auxiliary board which regulates the power and give Pulse-width-modulated (PWM) outputs for motors, a number of Analogue-to-digital ports and a number of digital input/output ports. The stack-able nature of the boards allows the configuration to be changed easily.

The microcontroller uses the analogue to digital converters built into a PIC16F873A chip (which also provides PWM outputs for used for motor control). The PIC, has 5 analogue inputs each of 10Bit resolution however when read via the robot link library this is rounded to 8bit accuracy. A simple 5v line is used as a reference, e.g.

$V_B = 256.\frac{V_{IN}}{5}$

if more accurate values are required a separate reference (<5v) needs to be fed to one of the other channels and the results scaled accordingly.

The connections are made via a 15 pin D socket with pin allocations.

Pin		Pin		
1	ADC 4	9	-15v	Not Normally Used
2	n/c	10	+15v	Not Normally Used
3	ADC 3	11	5v	
4	ADC 2	12	5v	
5	ADC 1	13	GND	
6	ADC 0	14	GND	
7	n/c	15	GND	
8	n/c			

There are a number of other interfaces on the microprocessor board and the auxiliary boards which give access to digital i/o and motor drives, in addition there is a LED array which shows the status of the system. The function of these indicators is described in the Software section SD.5.

ED.1.1 I^{2}C Bus

The sensor electronics on the robot communicate with the on-board microcontroller via a fast serial communication link, known as the I2C (Inter-integrated circuit) bus. The I2C bus was designed for serial communication between integrated circuits over short distances, such as on a PCB. The principle is that each IC on the bus has a unique address and may operate in either a master or slave mode. Data transfer is initiated by the bus master, of which there can only be one at any given time. The bus master transmits the address of the device with which it wishes to communicate, together with a single bit denoting the direction of the data transfer. The required number of bytes are then transferred. The limiting factor for the bus is the total capacitance which is connected to it. As it was designed for use on a PCB, care should be taken to minimise cable runs.

There are a number of commercially available devices for connection to the I2C bus. Amongst these are the Philips PCF8574 and 8574A, each of which is an 8 bit input/output port (see datasheet - available from the parts catalogue). Each of these device types can be selected by one of 8 possible addresses, thus providing ample capacity.

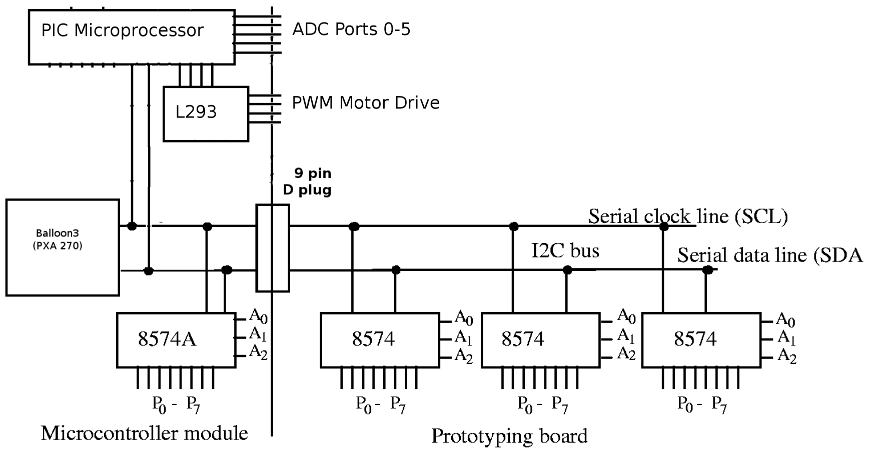


Figure 9: I^{2}C bus circuit

For this application there is only one bus master, the Balloon 3 microcontroller, which initiates all bus data transfer. The PCF8574(A) has 3 address pins (A2,A1,A0) for its 8 possible addresses. The actual address that the micro-controller must use is formed by a base address unique to the PCF8574(A), offset by the value on the address pins. All ports internal to the microcontroller module use the 8574A chip, so all 8 addresses in the 8574 are available for the prototyping board.

Each bit can be allocated to a specific function in the mobile robot interface. For example, the infra-red sensor port can use one bit as an output to switch the infra-red circuits on and off and four as inputs to receive the state of each of the four sensors. An input bit will be required for each microswitch and an output bit for each actuator. (See Section SD.1.1)

ED.2 Electrical Components

ED.2.1 Logic Family

The integrated circuits provided are CMOS circuits in the high-speed HC range. They are pin-compatible with the standard 74 series logic (originally TTL) family. Some members of the HCMOS family derive directly from the earlier 4000 series CMOS family and this is indicated in the code. General information on HCMOS and on the individual circuits in the electronics kit is contained in the datasheet. Additional data on the HC family is available on request.

The CMOS version of the IC 555 timer circuit (ICM7555) is provided.

ED.2.2 Discrete Components

Resistors and capacitors are listed in the Electronics kit and other values are available on request. The resistors are 1/4W and 5% tolerance. The capacitors are 20% tolerance. An n-channel enhancement mode power FET is available as is a double pole changeover relay (DPCO). The characteristics of both devices are given in the datasheet.

ED.2.3 Infra Red Emitter/Detector Assemblies

The infra red emitter/detector assemblies have been selected to provide the basic means of route detection. They may also be used to provide a means of obstacle detection which does not require a collision with the obstacle. The assembly contains a combined infra-red emitting diode and photo-transistor detector module. The photo-transistor responds to radiation from the diode when a reflective object is placed within the field of view of the sensor. (Standard module, see datasheet in the parts catalogue). When light is detected (e.g. reflected from the white line marking the route) current flows in the photo-transistor.

The emitting diode should be connected to the 5V supply with a series resistor to set the diode current. Similarly, a resistor should be connected in series with the transistor and the voltage across the resistor used as a measure of the current. This voltage may be converted into a logic signal using a Schmitt trigger circuit. A test box with a range of resistors is provided for experiments on the sensors.

There is a possibility of cross-talk between adjacent sensors, that is, light emitted from one sensor unit may also be detected by the adjacent unit. If this is a problem, a circuit using sequential excitation of the sensors may be needed to eliminate the effects of cross-talk. The switching function for sensor excitation can conveniently be obtained with the infra-red emitting diode (and a series resistor) as the load in a common-source FET transistor circuit. Sequential excitation of the sensors may also be used to reduce the average supply current to the sensors.

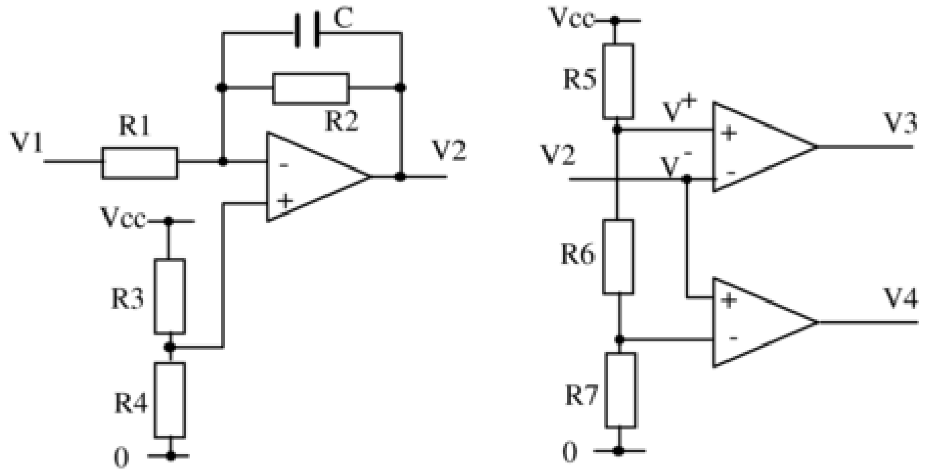
The states of two or more infra-red sensors may be combined by logic circuitry to detect certain features of the robot's environment. Microswitch states may be used in a similar way. While these combinations can be defined in the software, there may be speed and other advantages in using hard-wired logic with regular interrogation, or polling, of the logic circuit from the software. For example, the simultaneous ON condition of two IR line-following sensors could be used to detect a white line perpendicular to the line being followed. An AND gate plus a latch can be used to detect and hold the condition until it is next polled by the processor through the I2C bus. The latch must then be reset, either using an output signal from one of the I2C bus interface devices or by arranging to clear the latch when it is read.

ED.2.4 Detection/Identification Circuits

The technique used to detect and identify a task parameter will depend upon its particular characteristics. Suggested methods appropriate to your task are available as a separate handout from demonstrators.

Having detected the signal it is often necessary to be able discriminate between one of two or three states of the signal.

A circuit to discriminate between two input levels can be implemented (a) as an analogue comparator using a Schmitt trigger circuit or an op-amp with no feedback or (b) in software following analogue to digital conversion. (The output from an op-amp comparator will be (nearly) Vcc for V+ greater than V- and (nearly) 0 for V+ less than V-.) An example of an op-amp comparator circuit which can distinguish three input levels is shown in Figure 10(b)



(a) Mean value circuit (b) Three-level discriminator circuit.

Figure 10: Op-amp circuits

An MC33172 dual op-amp IC is provided (See Parts catalogue for the datasheet). Inputs to unused amplifiers should be connected to 0V. You may find that setting up voltage values for level discrimination etc is easier if you use variable resistors in combination with fixed resistors.

ED.2.5 Microswitches

The switch supplied is a single pole, double throw switch. The contact specifications are:

Contact rating	250V (ac) 30V (DC), 5A resistive load
Operating force (max)	140g
Release force (min)	25g
Over-travel (min)	1.0mm
Pre-travel (max)	4.6mm
Differential (max)	0.4mm

The switch and lever of the microswitch need protection from excessive force and over-travel so care is needed in the choosing the method by which the switches are mounted on the robot. The switch state must be converted into a well-defined logic signal. This can be achieved by a bistable circuit (conveniently formed using two 2-input NAND gates) or a Schmitt trigger circuit. The switching action can produce a series of pulses as the contacts bounce and the circuit chosen should prevent any unwanted outputs as a result of this effect. The states of two or more microswitches may be combined using logic circuits as described in the section on the IR sensors above.

ED.2.6 Actuators

The electrical characteristics and pin configuration of the solenoids in the pneumatic actuators are given in the section MD.4.1. The nominal rating of the solenoid coils is 5V, 1.2W (240mA), but the actual current consumption should be confirmed by measurement.

ED.2.7 LED Indicators

The aim of the LED indicators (Standard style, see Electronics Data Book) is to give an indication of the status of the sensors on the robot. Their function should be defined at the circuit design stage. You will need to consider the current requirements for these indicating LEDs and the value of the current limiting resistor to be connected in series with each diode.

ED.3 Power Distribution

Fig. 11 illustrates how the power from the 12V PSU is distributed to the microcontroller module, drive motors and the prototyping PCBs.

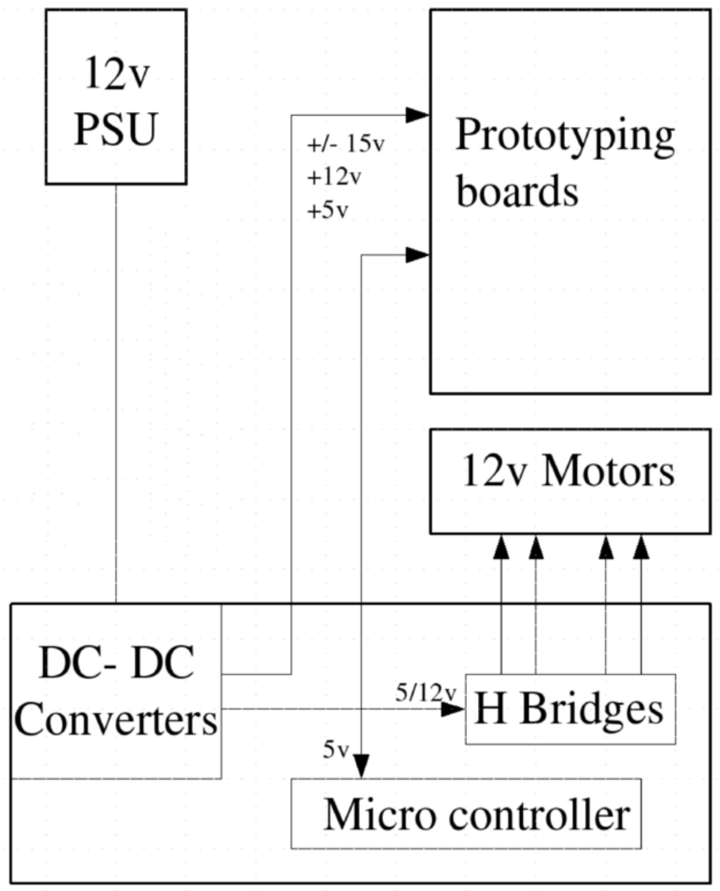


Figure 11: Power distribution to the microcontroller, drive motors and PCBs

ED.4 DC Motor/Gearbox Units

You may use up to three motor/gearbox units on your robot. Two final drive ratios are available: 20rpm and 40rpm and you may use two 20rpm and one 40rpm or one 20rpm and two 40rpm units. You may not have all three units with the same final drive ratio.

Two motor/gearbox units (either 20rpm or 40rpm) are supplied with the mechanical kit. For the third motor (or any gearbox change required) see a demonstrator in the EIETL.

Driving the motors

The microcontroller unit has four pulse width modulated (PWM) output ports of which only two are generally available for controlling the two robot driving motors.

More information on the L293 H-Bridge driver is available in the datasheet in the parts catalogue.

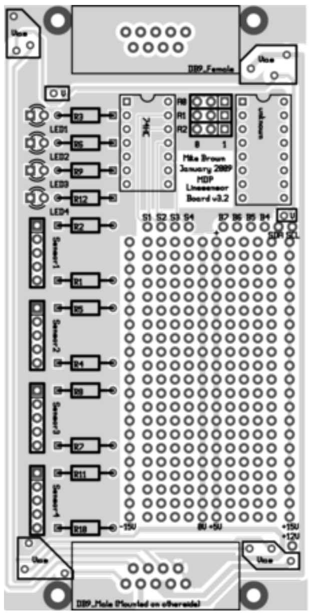
ED.5 IDP Pic module

A simple pic module is available that can be used to generate a simple multiprocessor system or as a simply programmable logic block. It uses a PIC12F1840 which has 5(6) op/ip, 4 of which can be configured as 10 bit ADCs if required. The module is progammable using a simple flow diagram language which automatically generates commented assembler. There is a IDP limit of a maximum of 35 flow diagram function blocks in any programme.

If a team wishes to use a module please discuss the application and obtain permission for it use a demonstrator before undertaking detailed design.

ED.6 Printed Circuit Boards

ED.6.1 Line Sensor - Prototyping PCB Configuration


(a) Component side

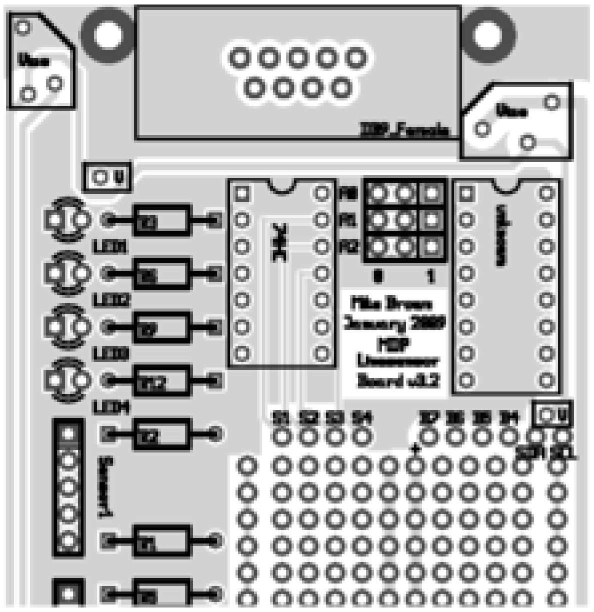

(b) Component Layout

Figure 12: Basic Line Sensor Prototyping PCB

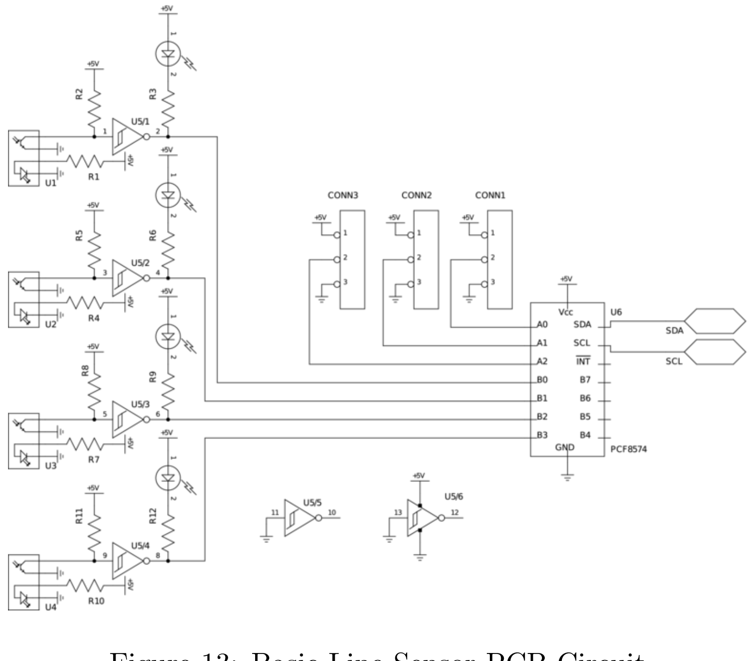


Figure 13: Basic Line Sensor PCB Circuit

Voltage source

5v and 0v are available from tracks that run on the solder side of the boards. (± 15 v is available from the D plug pad area)

Port address

The port's address is defined by the chip address lines A0 to A2. To set the address use the links (conn1 - conn3), linking the pins nearest to the 8574 sets the input to 5v, linking between the middle pin and remote pin sets the input to 0v.

ED.6.2 Prototyping PCB Configuration

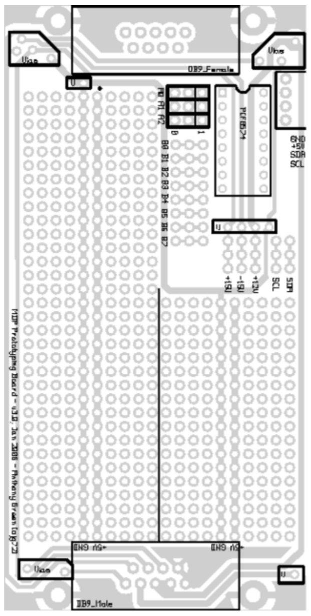


Figure 14: Prototyping PCB, component side

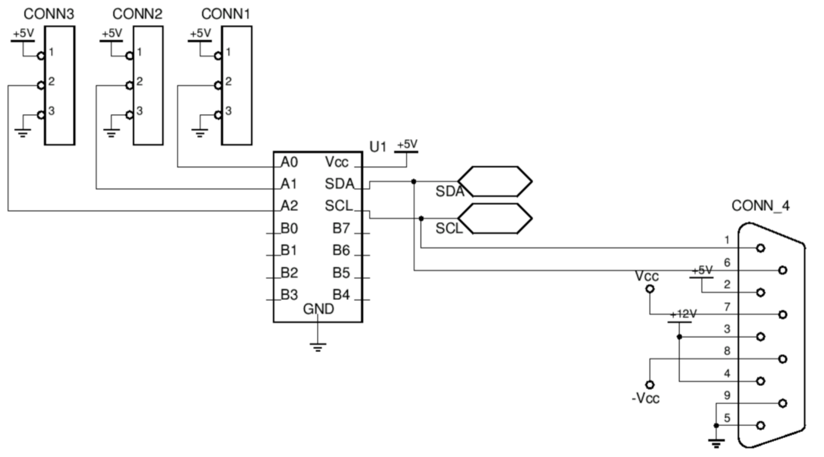


Figure 15: Circuit of Prototype PCB

Voltage source/ Port addressing

The available voltages are identical to the line sensing board, see above. The setting of the 8574 address is also done in a similar manner to the line sensing board.

IB Integrated Design Project

Mobile Robot - Software Design

S.1 Introduction

This section describes the software part of the Mobile Robot Project. The software provides the overall control for the robot and, in this project, can either run in general purpose workstations of the type used for other programming activities in the course or be downloaded into the on-board microprocessor. The former environment facilitates software development and is hence commonly used in the development phase of projects which have complex software components. The latter is more suitable for late prototype and, of course, production versions of the system.

As in the other tasks in the project, a kit of parts is provided. In this case, the kit consists of a set (a software library) of datatypes and functions which can be used by a C++ program to control the robot. The core of this library is a datatype (`robot_link`). There are different implementations of this for the two environments. For programs running in the workstation, the library controls the sending of three byte command messages from the workstation over a wireless network to the microprocessor in the robot (see the **IDP Technical Databook** for further details). This microprocessor interprets these instructions, produces signals to control the hardware and sends back three byte response messages to the software in the workstation. For programs running in the microprocessor, communication with the hardware is direct. This communications mechanism is fully provided by the software library and the details, including whether messages are sent over the network or not, are unimportant to the programmer (one of the benefits of having software components). The section SD.7 contains, for background information, some of the additional details of the implementation of the library.

A number of initial experiments are outlined which provide useful design data, familiarity with the programming environment and the library, and which calibrate some of the robot functions needed later in the project. Following this, the software development merges with the rest of the project and the aim becomes that of solving the overall robot design problem described in the Problem Statement.

Guidance is given on Software Engineering techniques, some additional features of the C++ programming language, and on aspects of report writing specific to the software part of the project. This guidance only suggests a general framework, there is a great deal of scope for individual creativity!

S.2 Methodology for Producing Software Systems

In the last thirty years, a new branch of Engineering, Software Engineering, has grown up. This, like other branches of Engineering, has a set of methodologies for designing and building (or implementing) systems. As in other branches of Engineering, one of the central ideas is that of decomposing a large difficult problem into a set of simpler sub-problems, these sub-problems may then be further decomposed into yet smaller sub-problems until a level of complexity is reached at which the solution becomes relatively trivial. The sub-problem solutions are the components of the complete solution and can be assembled to form a solution to the overall problem. Some of these components may pre-exist or may be useful at a later stage as parts of some other system; for example, components of the programs written during the initial experiments in this project are likely to be reusable as low level components in the final control program.

In the context of Software Engineering, the components are program elements like subroutines and functions. Associated with these are the datatypes and data objects on which they operate. Some methodologies (functional) place the principal emphasis on the program elements, others (object oriented techniques) on the data. Often the former approach is most suitable for the top levels (i.e. first levels of decomposition); whereas for the lower levels it may be easier to think in terms of the data first, then what operations are required on it and then to contain both within an abstract datatype (e.g. a C++ class - see section SD.1.3). For this project, the low-level components which are provided are C++ classes but the code to be written can follow either methodology.

Another important design issue is ensuring that the software components fit together. This typically involves specifying what data each requires as input and what output it produces. The clearest specifications arise when the input is in the form of parameters to a procedure or function and the output (if any) is the value returned by a function. C++ classes may be useful for grouping related data and operations, and for limiting the number of separate objects which need passing around. Global data (see section SD.1.4) should be used with care.

The production of a software system can be split into a number of phases –
The Software Lifecycle:

> **Requirements Definition** – define what the system is required to do.
> **System Design** – design the overall system structure.
> **Component Design** – design the individual components.
> **Implementation** – write the software which implements the component design.
> **Integration** – fit the components together and to other non-software parts of the system.
> **Testing** – making sure it works reliably (including its reponse to unusual or erroneous inputs).

These phases generally overlap and may not be entirely sequential, eg: component testing before integration; and iteration back from a later phase to an earlier one. Some iteration of this kind is expected but large loops back (e.g. testing revealing mistakes in the requirements definition) are clearly to be avoided if at all possible! Throughout the process, there is a further very important activity: the production of **documentation**.

Proper application of these techniques should result in software that is written and documented in such a way that it can subsequently be maintained and developed, not necessarily by the original authors. Bearing this in mind should help you to decide how successful you are being in producing good software.

When designing software, you should also bear in mind other engineering techniques which you have learnt. For example, the sequence of actions performed by the robot could be modelled as a state machine (IA Digital Electronics course) and the software could then be designed around this model.

S.3 Introduction to the use of the Software Library

The software library contains off-the-shelf components to control the robot and to provide some support functions. The facilities provided by the library, based around the class `robot_link`, are discussed in detail in sections SD.2, SD.3 and SD.4.

These describe the complete set of routines available (not all are required for the project; for example the instructions to control the ultrasonic sensors are only needed if these are being used).

In addition, section SD.1 provides an introduction to some aspects of the C++ language (like bit manipulation and using classes) which are not covered in the IA Computing Practicals but which are likely to be useful for this project.

S.3.1 A Simple Example Program

The following program shows simple use of the robot software library and illustrates some of its essential elements. Note the use of the dot operator ('.') to access the `robot_link` member functions in the same way as accessing structure data (Section SD.1.3 describes this in fuller detail).

```
#include <iostream>
using namespace std;
#include <robot_instr.h>
#include <robot_link.h>

#define ROBOT_NUM  33                        // The id number (see below)
robot_link  rlink;                           // datatype for the robot link

int main ()
{
    int   val;                               // data from microprocessor
    if (!rlink.initialise (ROBOT_NUM)) {     // setup the link
        cout << "Cannot initialise link" << endl;
        rlink.print_errs("    ");
        return -1;
    }
    val = rlink.request (TEST_INSTRUCTION);  // send test instruction
    if (val == TEST_INSTRUCTION_RESULT) {    // check result
        cout << "Test passed" << endl;
        return 0;                            // all OK, finish
    }
    else if (val == REQUEST_ERROR) {
        cout << "Fatal errors on link:" << endl;
        rlink.print_errs();
    }
    else
        cout << "Test failed (bad value returned)" << endl;
    return -1;                               // error, finish
}
```

In this program, the header files defining the robot instruction set (`robot_instr.h`) and the software interface for the link (`robot_link.h`) are included.

The program initialises the link, sends a test instruction and checks that this returns the correct test result from the robot. This program could thus be run on a workstation to verify that the link and the microprocessor in the robot are operating correctly. The ID `ROBOT_NUM` is the number on the network interface card plugged into your microprocessor. This will normally correspond to your group number within the block, e.g. for group M206 it would be 6. A version of this program to be downloaded and run on the microprocessor would not need `ROBOT_NUM` information since it communicates with the local hardware (see section S.3.3 below).

S.3.2 The Workstation Programming Environment

For programs to be run on the workstation, this part of the project mainly uses those facilities familiar from the Part I Computing Practicals, e.g. the IDPgeany program for compiling programs. Programs written during the project can be compiled using the facilities provided by the 1BRobotgeany desktop environment which is accessed via the "Applications" icon at the bottom of the Teaching System screen and then clicking on the 1BRobotgeany option in the "All CUED Applications/2nd year" section. This environment has desktop icons similar to those for the 1A and 1B Programming Classes plus, project management software and a link to the IDP Web site.

If you need help remembering how to use this environment, the online help, the Introduction to the Teaching System and C++ links may be useful.

A special shared group file space is set up for each team for the project. Within this there are folders (or directories) for each team member (named with their login identifier) and a Common folder:

- Each team member can read and write files in their own folder and the Common folder; and read files in one anothers folders.
- There is an icon on their desktop for a team members own shared folder as well as the usual icon for their private home folder (for unshared files). Double clicking on this shared folder launches the file browser providing access to the folder and to the others in the shared group filespace.
- There is a link created in each team members private home folder named idp shared pointing to the teams shared folder. This is useful for accessing this folder from programs like a terminal window which start up in the home folder.

S.3.2.1 Geany

Just drop the source files all at the same time onto the icon and click on Make. Alternatively, drop a folder onto the icon (in which case all the *.cc, *.C, and *.cpp files in the folder will be used). Here are some frequently asked questions

1. When I use the geany icon, the compiler complains about the robot commands - make sure you're using the IDPgeany icon, not the geany icon left over from the 1st year. The IDP icons appear when you start a session using the 1BRobotgeany option
2. When I use the geany IDP icon, Build doesn't work - you need to use Make in the Build menu: clicking on the Brick icon isn't enough.
3. What files do I drop into the compiler icon? - all the files that comprise the project except files that you have #included. You need to drop them in all at the same time. Alternatively, drop a folder onto the icon, but make sure the folder only contains project-related source files
4. I've created a file in geany, but I can't compile it - Quit from geany, then drop the source file(s) into geany
5. I've been developing code on windows. Now I've copied it onto the CUED machines it doesn't compile - use dos2unix to convert the file's line-endings.
6. I have several independent source files in geany. Flicking between them amd using Make doesn't work. - This version of geany (unlike the 1A Mich version) assumes that the files initially dropped onto it form a single program. Close geany and load just one of your files in.
7. The Execute ("gearwheel") button does nothing - this can happen if you're running a program that's in a folder you're not allowed to create files in.

S.3.2.2 Other IDEs

Several other Integrated Development Environments are available on the system. Note that

- Compiling needs CCSWITCHES="-I/export/teach/1BRobot"
  and
  LIBS="-L/export/teach/1BRobot"
- Cross-compiling needs
  CCSWITCHES="-I/usr/arm-unknown-linux-gnueabi/include -I/export/teach/1BRobot"
  and
  LIBS="-L/usr/arm-unknown-linux-gnueabi/lib". The compiler is arm-linux-gnueabi-g++

S.3.3 The Microprocessor Programming Environment

In addition to running programs on the workstation, it is also possible to cross-compile them and download them to run on the microcontroller directly. The processor in the microcontroller does not use the same machine code instructions as the workstation. It uses an Intel XScale PXA270 processor that incorporates an ARM microprocessor core. Code compiled to run on a workstation will not run on the microcontroller, instead you must use an ARM cross-compiler.

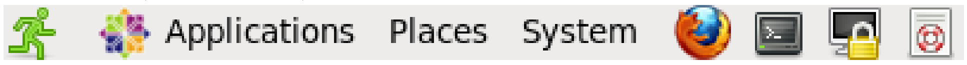
S.3.3.1 Cross-compiling, Downloading and Running Programs

The following very simple example shows how to cross-compile and run a program on the microcontroller. Suppose the program is in a file `hello.cc` containing

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello World" << endl;
}
```

To cross compile and link the above program, drag the hello.cc file to the IDPgeany-arm icon (rather than the normal IDPgeany icon) on the desktop, and click on the Make button. This will produce an executable file named hello.arm (.arm distinguishes it from the executable hello which IDPgeany would produce to run on the workstation). Next

- Copy the file to the robot. To do this you need to use the unix command line to run a program called scp (SecureCoPy). Use the Terminal icon in the task ba
  

  to give you a Unix Terminal window with a command line.
- A short Unix crib is online, telling you about 3 useful unix commands: ls (to list files), pwd (to see which directory (aka folder) you're in) and cd (to change directory). It's useful to go to the directory (aka folder) where your program is. You can do this by typing

  ```
  cd directory-name
  ```

  but there's a short cut. Using the File Viewer, display the folder where hello.arm is (e.g. abc12/IDP-Shared/abc12/python2 folder). In the Terminal window, type cd followed by a space, then with the mouse drag the folder's name (in this case python2) into the Terminal window.

  ```
  cd '/users/s/abc12/IDP-Shared/abc12/IDP/python2'
  ```

  Press the Enter key to change directory.
- Now use the scp command from the Terminal window, replacing 100 in the example below by the number on the WiFi card.

```
scp hello.arm team@wlan-robot100.private:hello.arm
```

  - *scp* is the program you're running. It's not in the current folder (it's a system command) so you don't need an initial ./
  - *hello.arm* - the name of the file you're copying over.
  - *team* - the name of the user on the robot. It's always team
  - *wlan-robot100.private* - the name of the robot you're using. Don't use 100. Use the number on the WiFi card.
  - *hello.arm* - the name the file will have on the robot. It needn't be the same as the original file-name

  scp might print messages about authenticity. Don't worry - just say yes. It will then prompt for the team's password: enter the password printed on the side of the microcontroller box. When you type the password nothing appears on screen.

- Run the program. First log into the robot, replacing 100 in the example below by the number on the WiFi card.

  ```
  ssh team@wlan-robot100.private
  ```

  After typing the password correctly you'll be logged into the robot and you'll be given a Unix prompt . You can type ls to list the files on the robot. You should see hello.arm there. To run it, type

  ```
  ./hello.arm
  ```

  (the '.' means the 'current folder' - by default, the system doesn't look in the current folder for programs, so you need to make it look there)

You can either logout from the microcontroller at this point to return to work on the workstation or, perhaps more conveniently, leave this terminal window logged-in to the microcontroller and open another on the workstation to do subsequent program development, compilation and downloads.

S.3.3.2 Cross-compiling robot control programs

This follows much the same method as above. In the same way that IDPgeany does for the workstation, IDPgeany-arm knows about the appropriate header files and libraries for programs to be run on the microcontroller and can handle multiple source files as well as just one. The only difference in use between IDPgeany-arm and IDPgeany is that the output executable from the former is name.arm (and will only run on the ARM chip) whereas for the latter it is just name. Apart from using IDPgeany-arm rather than IDPgeany, the only other change normally required is to the initialisation of the robot link object (see later for details). When running on the workstation, the network interface card number is required; when running on the microcontroller, none should be specified to indicate that communication is local.

This can most easily be achieved by using conditional compilation (see section SD.1.6) and using whether or not `__arm__` is defined to distinguish between the two compilers and hence select the correct version. Thus for the program in

S.4

S.3.1 above, the `rlink.initialise` line is replaced by

```
#ifdef __arm__
    if (!rlink.initialise ()) {              // setup for local hardware
#else
    if (!rlink.initialise (ROBOT_NUM)) {     // setup the link
#endif
```

S.5 Tasks to be performed

S.5.1 Formulating the Software Requirements

You should start to define the functionality of your software at an early stage both to ensure that the software you write can meet the evolving system specification and so that you can start to produce components of it (eg procedures and suitable datatypes) as soon as possible. Bear in mind when designing and writing your software that the design of other parts of the system may have to change due to unforseen problems and that the software should be easily adaptable to these changes.

One aspect of the design which will need early thought is the line following algorithm since this will depend on such factors as response time and sensor placement.

S.5.2 Initial Tasks and Experiments

See the **Week 1 Tasks and Exercises** in the IDP manual

S.5.3 Other Tasks

Alongside these tasks, the other sub-teams will be producing their subsystems and may need you to produce test programs to help to test these out. For example at some point you will be able to replace the test chassis by the proper one. You should discuss these requirements as a team.

Towards the end of the above tasks, you should be in a position to start the Software Lifecycle for the complete final program which will control the robot when it is performing the task described in the Problem Statement. See the Design Acceptance section in teh logistics document for guidance on what the design and implementation documentation should contain and the earlier discussion (S.2) of software engineering and the Software Lifecycle.

Testing should take place alongside the mechanical and electrical systems as each is refined to produce the final working robot. Initially this testing is likely to comprise component testing of particular software modules, e.g. line following, against specific electrical and mechanical subsystems before the final complete system testing is undertaken. It is important to aim to ensure that any problems are discovered as early as possible in the development process.

SD.1 Additional C++ Language Features

This section provides an introduction to some aspects of the C++ language needed for this project which are either not covered at all or only very briefly in the IA Computing Practicals. The on-line help system information on `C++` may also be useful.

SD.1.1 Bit Manipulation

As well as being used to represent numbers, integer (`int`) and related datatypes may also represent a set of bits (e.g. signal values for an input or output signal from a chip). These bits can be manipulated using the C++ `bitand`, `bitor` and `xor` operators (or their equivalents `&`, `|` and `^`) which are analogous to the corresponding instructions in a typical microprocessor instruction set, eg

```
    int  param;                 // a  value
    const int bit0 = 0x01;      // '0000 0001' individual bits
    const int bit5 = 0x20;      // '0010 0000' expressed in hexadecimal

// ... code to setup the link etc not shown

    param = bit0 bitor bit5;        // '0010 0001' set to 21 hex (33 dec)
// send command to set bits 0 and 5 in I2C chip at address 4
    rlink.command (WRITE_PORT_4, param);

// get value from I2C chip at address 1
    param = rlink.request(READ_PORT_1);
    if (param bitand bit5) {         // check if bit 5 is set
        // ... and act accordingly
    }
```

There are also C++ integer operators for **left** (`<<`) and **right** (`>>`) shift, e.g.

```
    bit5 = bit0 << 5;
```

sets `bit5` to 20 hex as above. Note that these operators are equivalent to multiplying/dividing by 2^N, e.g.

```
    bit5 = bit0 * 32;           // is the same as bit5 = bit0 << 5;
```

(since $2^5 = 32$) but the former (`<<`) is usually clearer than the latter when thinking in terms of sets of bits rather than numbers. The next section addresses a similar issue of clarity of programming style.

SD.1.2 Hexadecimal Values

In the above, the prefix `0x` before a number indicates hexadecimal notation, e.g. `0x10` is `0001 0000` in binary and `16` in decimal. This is probably most useful when expressing values corresponding to one or more bits being set, for example it is clearer that two bits are set in `0x1020` than if the same value were expressed in decimal (`4128`) but other than making the program more readable it does not make any difference which form is used.

(Note that values with a leading zero (but no following **x**) are octal, ie base 8. Their only real importance to this project is as a possible cause of error when expressing a decimal value (e.g. `0100` is `64` not `100`!).

SD.1.3 Classes

C++ *classes* are an extension of the structured datatype *struct* in which functions as well as data may be declared within the datatype as members. These *member functions* may be accessed in the same way as a data member, for example `object.func(param)`.

In addition to having member functions, the members of a class may either be *public* (that is part of the accessible interface to the class) or *private* (part of the hidden implementation of the class and accessible only to the member functions). This is important from a Software Engineering standpoint as it means that the detailed implementation of a class may be changed without affecting programs which use the class, since these rely solely on the public interface.

The class `stopwatch` (described in section SD.4) provides a simple example of a class. `robot_link` (in section SD.2) is a more complex example.

```
class stopwatch {
public:
    stopwatch ();
    void start ();
    int read ();
    int stop ();
private:
    struct timeval base_time;
    bool  running;
};
```

In this case, the member functions are all public but the data members are private to the implementation of the class. Providing access to the class's data only via functions is a commonly employed technique to ensure that the data values cannot be set to inconsistent values and/or that the detailed encoding of the data does not need to be understood by the rest of the program.

SD.1.3.1 Constructors and Destructors

A member function with the same name as the class acts as a **Constructor** which is called *automatically* when the object is created, ensuring that suitable initialisation of the object is performed. One with the same name preceded by a ~ is a **Destructor** and is called automatically when the object is destroyed. The class `robot_link` has both a Constructor and a Destructor; `stopwatch` has only a Constructor.

SD.1.3.2 Overloading of Member Function Definitions

Member functions can be *overloaded*, that is there can be more than one function with the same name but distinguished by the type of parameters given. The different forms of `initialise` for `robot_link` are an example of this.

SD.1.4 Global Data

Variables declared within a function body are local to that function, ie they can only be referred to by code in that function. Variables declared outside any function body are **global**, that is they are visible to all functions from that point on. Global variables can be very useful as a way of avoiding having to pass values between functions as parameters or returned values. However incautious use can lead to programs which are very hard to understand and maintain. A useful pragmatic guideline is to have as global data only objects whose value is set by the top level functions (e.g. `main`) and read by several of the lower layers. The problems arise when global data is modified by many different parts of the system.

SD.1.5 Multiple Source Files

It is frequently convenient to split the source code for a program into more than one file, e.g. so that different people can work on different parts of the program. This also makes it possible to arrange that data or subroutines are private to the file, i.e. hidden within the module so formed, much like the private section of a class. Where the declaration of a type, a function or a variable should be visible to more than one source file, this can be achieved by putting the declarations in a header file (customarily given the suffix `.h`) which can then be included in each of the source files which need access to the declarations. This ensures that all the source files use consistent declarations.

The example in below shows how to include various system-provided header files.

```
#include <iostream>
using namespace std;
#include <robot_instr.h>
#include <robot_link.h>
```

The syntax to include files in the current directory is similar but uses quotes (`"..."`) rather than angle brackets (`<...>`) to enclose the name. E.g. consider a source file (`mysource.C`) containing:

```
#include "myheader.h"

int aglobal = 3;       // global variable initialised to 3
static int alocal = 0; // variable local to all the functions in this file

int myfunc (int arg)   // a global function which manipulates alocal
{
   alocal += arg;
   return (alocal);
}
```

and the header file (`myheader.h`) with the global declarations:

```
extern int aglobal;
int myfunc (int arg);
```

This header file is included in any source files which refer to these global functions and variables (simply by using their names `myfunc` and `aglobal`). The header file is also included in `mysource.C` which defines the function `myfunc` and the variable `aglobal` so that the compiler can check that the declarations and definitions are consistent.

The definition of a function specifies what it does, the definition of a variable allocates space in memory for it and optionally sets an initial value. Definitions of functions and variables must therefore occur in only one source file, whereas the declarations can be included in many. Type declarations, e.g. the class `robot_link` in `robot_link.h`, appear in the header file but the code defining any associated methods are in a separate source file implementing the class.

If you're having trouble, experiment with some minimal files.

Below is a multi-file version of the earlier test program

```
// header.h
extern robot_link rlink;
int request();

// one.cc
```

```
#include <robot_instr.h>
#include <robot_link.h>
#include "header.h"

int request() {
return rlink.request (TEST_INSTRUCTION); // send test instruction
}

// two.cc
#include <iostream>
using namespace std;
#include <robot_instr.h>
#include <robot_link.h>
#include "header.h"
#define ROBOT_NUM 33   // The id number (see below)
robot_link rlink;      // datatype for the robot link

int main ()
{
int val;                              // data from microprocessor
if (!rlink.initialise (ROBOT_NUM)) { // setup the link
  cout << "Cannot initialise link" << endl;
  rlink.print_errs("  ");
  return -1;
}
val = request();
if (val == TEST_INSTRUCTION_RESULT) {   // check result
  cout << "Test passed" << endl;
  return 0;                            // all OK, finish
}
else if (val == REQUEST_ERROR) {
  cout << "Fatal errors on link:" << endl;
  rlink.print_errs();
}
else
  cout << "Test failed (bad value returned)" << endl;
return -1;                          // error, finish
}
```

Dropping one.cc and two.cc together onto the IDPgeany icon creates a file called Makefile behind the scenes. Clicking on geany's Make menu item will produce a program called two (named after the file containing the main function).

Note that it's not usually a good idea to

- Use #include to include source files
- Create variables in header files

SD.1.6 Conditional Compilation

It is sometimes convenient to be able to select whether certain sections of code in a source file are compiled or not. C++ provides `#ifdef SYMBOL` (if SYMBOL is defined) and `#ifndef SYMBOL` (if not defined) and `#else` and `#endif` for this purpose. A typical example would be whether or not to include code intended for debugging purposes depending on whether the symbol DEBUG is defined, e.g.

```
#ifdef DEBUG
  cout << "Nearing junction three" << endl;
#endif
```

The symbol (e.g. `DEBUG` above) could be defined either by using `#define` or by a command line option to the compiler. The mechanism can also be used to test for symbols predefined by the compiler, for example the C++ cross-compiler for the microcontroller defines the symbol `__arm__` whereas the normal C++ compiler for the workstations does not. Thus to select appropriate code for use on the microcontroller or the workstation, use

```
#ifdef __arm__
  // code for microcontroller version here
#else
  // code for workstation version here
#endif
```

This mechanism is also useful to guard against multiple inclusion of header files. If your header files have the following structure

```
#ifndef MY_SYMBOL_H
#define MY_SYMBOL_H
// code here
#endif
```

(where the MY_SYMBOL_H part - the guard - is different in each file) then you won't accidently include a file more than once in a source file.

SD.2 Library - Mobile Robot Link

This section describes the low level routines in the datatype `robot_link` for setting up a network connection, sending instructions over it and handling errors. There is a one-to-one correspondence between a network connection and a data object of type `robot_link`; that is, a single data object controls a single connection between a workstation and a robot controller.

The instructions are of two types:

1. **command**s which are instructions sent to the robot
2. **request**s which obtain information from the robot

The next section(SD.3) details these instructions. Section SD.7 gives more detail of how the instructions are communicated to the robot.

SD.2.1 The `robot_link` Datatype

The public part of the C++ class `robot_link` (minus some specialist features intended for other applications) is defined in `robot_link.h` as follows:

```
class robot_link {
public:
// Construction/Destruction
    robot_link();
    ~robot_link();
// Initialisation
```

```
    bool initialise();                      // connect on local computer
    bool initialise(int card);              // connect to network card N
    bool initialise(const char *host);      // connect to specified host
    bool reinitialise();                    // reconnect after error
// Input output
    bool command (command_instruction cmd, int param);
    int request (request_instruction);
// Error handling
    void clear_errs();
    link_err get_err ();
    link_err lookup_err (int n);
    void print_errs(const char *prefix);
    void print_errs();
    bool any_errs ();
    bool fatal_err ();
}
```

The initialisation and input/output functions set appropriate error values (see section SD.2.4.1) if an error occurs.

SD.2.2 Initialisation

SD.2.2.1 Object Creation

The Constructor (`robot_link`) and Destructor (`~robot_link`) provide basic initialisation of a `robot_link` object and orderly shutdown of the link when it is destroyed.

SD.2.2.2 Link Initialisation

The member function `initialise` sets up the network connection between the workstation and the robot's on-board microprocessor.

`bool initialise()` – initialises a connection locally, e.g. for when the on-board microprocessor is running the control program.

`bool initialise(int card)` – initialises a connection to the given numbered network interface card.

`bool initialise(const char *host)` – initialises a connection to the named host.

In each case a boolean value is returned indicating whether the initialisation has succeeded.

SD.2.3 Instruction Input/Output

Input/output involves the exchange of request and command instructions between the program and the robot microprocessor. Each instruction consists of an opcode (the request or command to be performed) and a parameter (the value associated with it).

The interface is provided by the member functions `command` and `request`:

`bool command (command_instruction cmd, int param)` – sends the command `cmd` with parameter `param`. It returns `true` if no error has occurred.

`int request (request_instruction)` – sends the request `req` and returns the value supplied by the microprocessor or the special value `REQUEST_ERROR` if an error occurs.

SD.2.3.1 Link Resynchronisation

`bool reinitialise()` – closes and reopens the connection. It is typically used when recovering from a link error (see section SD.2.4 below), eg

```
if (rlink.fatal_err()) {   // check for errors
    rlink.print_errs();    // print them out
    rlink.reinitialise();  // flush the link
}
```

SD.2.4 Error Handling

Link operations (initialisation and input/output) may fail. Codes representing these errors are stored in the class `robot_link` and these stored values may be retrieved in various ways for subsequent processing. In addition to the error codes, counts are kept of certain types of error.

SD.2.4.1 Error Codes

The type `enum link_err` defines the following error values:

`LINKERR_NONE` – special value returned by `get_err` when there is no error to report.
`LINKERR_COMMS` – a fatal (ie unrecoverable) communications error.
`LINKERR_READ` – a read error receiving data from the link.
`LINKERR_WRITE` – a write error sending data to the link.
`LINKERR_CHKSUM` – a checksum error in the instruction packet returned from the microprocessor.
`LINKERR_NOTINIT` – attempt to use the link before it is initialised.
`LINKERR_BADCARD` – the specified card number does not correspond to any in the list of known cards.
`LINKERR_BADHOST` – the specified host name does not correspond to that of any known computer.
`LINKERR_BADPORT` – invalid port number specified.
`LINKERR_SKT1`
`LINKERR_SKT2`
`LINKERR_SKT3` – errors setting up connection to remote computer.
`LINKERR_NOHOST` – the specified host could not be contacted.
`LINKERR_NOSERV` – the network service (`robot_link`) for this application is not running on the specified host.
`LINKERR_OVERFLOW` – too many errors to record. If too many errors occur, this value is stored and further logging of errors is stopped.

SD.2.4.2 Resetting Error Recording

`void clear_errs()` – Clears all recorded errors and error counts (as at initialisation of the link).

SD.2.4.3 Retrieval of Error Codes

`bool any_errs()` – returns true if any errors are stored in the error record.
`link_err get_err ()` – returns the next recorded error code or `LINKERR_NONE` if no more are recorded. This operation removes the record of this error.
`link_err lookup_err (int n)` – returns the $\texttt{n}^{th}$ recorded error or `LINKERR_NONE` if fewer than `n` are recorded (or `n` is less than one). This does not alter the error record.
`bool fatal_err()` – returns true if an unrecoverable error has occurred since the last time the error record was cleared (by `clear_errs` or `print_errs`).

SD.2.4.4 Printing Error Descriptions

`void print_errs(prefix)` – prints out all the recorded errors, one per line, prefixing each line with the character string `prefix`. The error record is cleared.

`void print_errs()` – as above but with no prefix string.

SD.3 Mobile Robot Instruction Set

The command and request instructions are described together, grouped according to the area of robot activity controlled. The examples assume that a `robot_link` object, `rlink`, has been declared.

SD.3.1 General I^2C Bus Input/Output Instructions

```
rlink.command (WRITE_PORT_0, v)
...
rlink.command (WRITE_PORT_7, v)
```

Writes the value `v` to the I^2C PCF8574 chip with the wired address indicated by the command opcode given as the first parameter. The concept of I^2C wired addresses for the PCF8574 is explained in section ED.1.1.

```
v = rlink.request (READ_PORT_0)
...
v = rlink.request (READ_PORT_7)
```

These return the value `v` read from the I^2C chip with the wired address specified by the command opcode. Note that the value read for a bit is low if *either* the external input is low *or* the value previously written to the bit is low – in other words, for a bit to be useful as an input it must first have a one written to it (as for bit 5 in the example in section SD.1.1).

SD.3.2 A to D Converter Instructions

`v = rlink.request (ADC0)`

...

`v = rlink.request (ADC4)` – These return the value `v` from the analogue to digital converter port (0..4) specified by the command opcode.

SD.3.3 Motor Instructions

```
rlink.command (MOTOR_1_GO, speed);
rlink.command (MOTOR_2_GO, speed);
rlink.command (MOTOR_3_GO, speed);
rlink.command (MOTOR_4_GO, speed);
rlink.command (BOTH_MOTORS_GO_SAME, speed);
rlink.command (BOTH_MOTORS_GO_OPPOSITE, speed);
```

The parameter `speed` for each of these commands is of sign-magnitude form: it ranges from 0 to 127 and the highest bit, if set, specifies the reverse direction. Thus, 153=128+25 gives a slow reverse. The relationship between forward/reverse and clockwise/anticlockwise depends on the gearbox ratio.

The first four commands refer to the single motor specified. The last two commands affect motors 1 and 2, in the first case rotating the wheels in the same sense for both these motors and in the second in opposite directions.

```
speed = rlink.request (MOTOR_1);
speed = rlink.request (MOTOR_2);
speed = rlink.request (MOTOR_3);
speed = rlink.request (MOTOR_4);
```

These return the current demanded speed for the relevant motor, encoded as for the motor-go commands above. This takes into account the effect of ramping (see below) but not mechanical load on the motor.

```
rlink.command (RAMP_TIME, t);
```

This sets the ramp period `t` for the motors. This should be chosen according to the friction between the wheels and the surface to obtain a good compromise between wheelspin and slow response problems. `t` ranges from 0 (no ramping) to 254 (slow ramping), with 255 specifying the default ramp period.

SD.3.4 Status Register Instruction

`stat = rlink.request (STATUS)` – returns the contents of the microprocessor's general status register. This contains the following bits, several of which correspond to the diagnostic LEDs (see section SD.5):

Bit	Function	
0	Communications error	(R)
1	I^2C Bus error	(R)
2	Emergency Stop Triggered	(R)
3	Emergency Stop Mode	
4	Robot Moving	
5	Ramp function enabled	
6	(unused)	
7	(unused)	

The bits marked (R) are reset on read.

SD.3.5 Emergency Stop Instructions

The emergency stop commands allow the robot motors to be stopped automatically, at a rate specified by the ramp time, when some condition or set of conditions is met. One of the most common uses is to stop the robot, when a microswitch closes, to prevent collision damage. A combination of bits being set high (specified by `Mh`) and/or low (specified by `Ml`) on the I^2C chip at address (`n`) may be used. `Mh` and `Ml` should be altered only while Emergency Stop Mode is disabled otherwise unpredictable behaviour may result. Both `Mh` and `Ml` are initialised to zero when the controller is powered up or reset.

`rlink.command (STOP_SELECT, n)` – enables Emergency Stop Mode, selecting the I^2C chip at address `n` to be polled continuously for the emergency stop condition. `n` is the I^2C wired address as described in section SD.3.1, except that specially a value of `n` with the most significant bit set (e.g. 128) disables Emergency Stop Mode. Emergency Stop Mode must, if required again, be explicitely reenabled after it has been triggered.

`rlink.command (STOP_IF_HIGH, Mh)` – the emergency stop condition will be triggered if any of the bits set to one in `Mh` is *high* in a value read from the I^2C chip.

`rlink.command (STOP_IF_LOW, Ml)` – the emergency stop condition will be triggered if any of the bits set to one in `Ml` is *low* in a value read from the I^2C chip.

Thus to cause an emergency stop when either bit 0 or bit 2 becomes high or bit 4 becomes low on the inputs to the I^2C chip at address 4:

```
  rlink.command (STOP_IF_HIGH, 0x05);
  rlink.command (STOP_IF_LOW, 0x10);
  rlink.command (STOP_SELECT, 4);
```

SD.3.6 General Instructions

`value = rlink.request (TEST_INSTRUCTION)` – This should always return the constant value `TEST_INSTRUCTION_RESULT` (as defined in `robot_link.h`). Reception of any other value indicates a communications error.

`rlink.command (SHUTDOWN_ROBOT, 0)` – This command causes emergency shutdown of the robot and it is returned to near initial state, eg the motors are stopped and most settable parameters are reset to their default values. The parameter value is ignored.

SD.4 Library - Support Routines

SD.4.1 Input/Output to Files

Information on input/output to files is provided in the sections on *Text Input and Output* and *Files* in the C++ chapter in the *Programming Databook*.

SD.4.2 The `stopwatch` Datatype

The public part of the C++ class `stopwatch` is defined in `stopwatch.h` as follows:

```
class stopwatch {
public:
    stopwatch ();
    void start ();
    int read ();
    int stop ();
};
```

SD.4.2.1 Initialisation

The Constructor (`stopwatch`) performs basic initialisation of a stopwatch; it is created stopped.

```
    #include <stopwatch.h>
    ...
    stopwatch  watch;
```

SD.4.2.2 Starting Timing

```
watch.start();
```

Causes stopwatch `watch` to start (or reset and restart) timing.

SD.4.2.3 Reading the Elapsed Time

```
etime = watch.read();
```

Returns the elapsed time `etime` in milliseconds since the `watch` was (re)started. If the stopwatch is not running, a value of zero is returned.

SD.4.2.4 Stopping Timing

```
etime = watch.stop();
```

Causes stopwatch `watch` to stop timing. The elapsed time (as for `watch.read`) is returned.

SD.4.3 Time Delay

SD.4.3.1 The `delay` Function

```
#include <delay.h>
...
delay (msecs);
```

`delay` (defined in `robot_delay.h`) provides a time delay of approximately `msecs` milliseconds.

SD.5 The Diagnostic LEDs

The eight LEDs have the following functions:

0	Red	Communications error
1	Red	I^2C Bus error
2	Yellow	Emergency Stop triggered
3	Yellow	Emergency Stop Mode enabled
4	Yellow	Robot Moving
5	Yellow	Ramp function enabled
6	Green	Communications active
7	Green	Watchdog (flashes while microprocessor is running)

SD.6 Sensor Simulator PCB

This circuit board enables software routines which interface with I^2C chips to be tested before the electronics and mechanical systems have been produced. The board has a single PCF8574(A) I^2C chip (see datasheet in the Parts catalogue for further details) whose wired address may be set to any 3 bit value by a set of switches. The chip has 8 input/output lines and another set of switches may be used to provide input data, and LEDs to indicate the output values.

SD.6.1 Board Layout

SD.6.2 Using the Board

The chip's address must be set *before* power is applied to the system (if the address is changed with power applied, the system may crash).

To use the chip for output, set all the data switches to 1 (the switches will otherwise override the chip's outputs). The LEDs will then indicate the values written to the chip's output port.

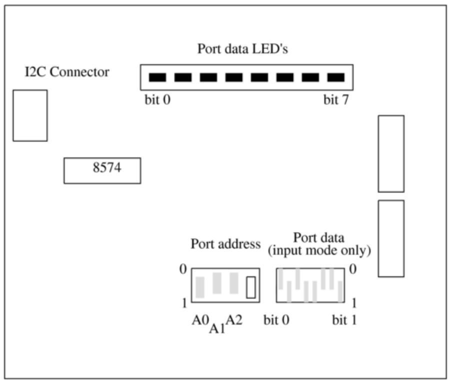


Figure 16: Sensor Simulation PCB

To use the chip for input, set the data switches to the required value and note that this value should also be indicated on the LEDs (on = 1). If the LEDs for bits which should be high are not on, this is likely to be because the corresponding bits in the chip's output port have not been set high and are overriding the switch value (see section SD.3.1). Port read operations will return this value.

SD.7 Details of the robot command interface

This section contains background material which may be of interest but is not essential to completing the project.

Communication between a workstation and the onboard microcontroller on the robot takes place via the Departmental Ethernet network. The workstations and cluster server are on one subnet and the robots on a special restricted-access wireless subnet. Traffic is routed between these subnets by the network infrastructure.

Instructions (commands and requests) sent from a C++ program and responses sent by the robot each consist of a three byte message: the command identifier, its parameter and a checksum. The robot never generates unsolicited messages, that is, each message from the robot is in response to a command or request from a program. Similarly all commands and requests result in a response message from the robot.

When the control program is run directly on one of the EIETL workstations (e.g. tw901) rather than a cluster server (e.g. tw900), the link is likely to have a slightly more predictable response time since these machines would not typically be running jobs concurrently for other users or needing to compete with other processes for access to the network.

The class `robot_link` illustrates the way in which implementation-specific detail can be hidden. The interface to a `robot_link` object is the same irrespective of the communication mechanism used, for example an earlier version of the IDP robot controller used a RS232 serial line rather than a network connection but had an almost identical software component interface (the differences being other versions of initialise() and some new error codes).

`robot_link` also illustrates how careful checking of types can help to prevent programming errors. For example, the `request` member function can only operate on a `request_instruction`.

K.1 Introduction

- See also on the IDP website for more details and datasheets.

K.1.1 Mechanical Materials/Components

Transmission components	**Per kit**	
Wheels, soft rubber tyre, 100 dia	2	4 available
Wheels, soft rubber tyre, 75 dia	2	
Castors, soft rubber tyre, 50 dia	2	
Either Large motor/gearbox 12v DC, 20rpm & 40rpm	Initially 2 of 1 speed	Three motors max, two of any one speed or size.
or Small motor/gearbox 12v DC, 18rpm & 40rpm	Initially 2 of 1 speed	
Wheel/motor connector kit	2 pairs	
Pneumatic components		
Pneumatic actuator (inc nut and clevis) 10 bore × 45 stroke	2	
Swivel connector (mounted on actuator)	4	
Pneumatic valve assembly	1	

Structural materials (available in the workshop)	**Per team**	
Mild steel sheet, 22swg (450×450 max)	1	Note: swg to mm conversions
Aluminium sheet, 1.5mm (200×150 max)	1	(swg = standard wire gauge)
Aluminium tube, 19 dia, 1mm wall, 150 long	1	24swg = 0.6mm
Aluminium rod, 8 dia, 150 long	1	22swg = 0.7mm
Aluminium angle, 12.5×12.5×1.6 (600 max)	1	21swg = 0.8mm
Mild steel tube (5/16 od × 21swg) 300 long	1	18swg = 1.2mm
Mild steel tube (3/8 od × 22swg) 300 long	1	16swg = 1.6mm
Polypropylene sheet 300×150×1.5	1	
Mild steel rod 4 dia 300 long,		
Mild steel rod 6 dia 300 long,		
Brass rod 6 dia 100 long,		
Brass hex 6 AF 100 long,		
Pneumatic materials		
Pneumatic hose, 1500mm max (from EIET Lab)	1	
Pneumatic 'T' connector (if required) from Demonstrators	2	

Fasteners and other materials (available in the workshop)

M2.5/M3/M4/M5/M6 nuts & washers

M2.5 × 12 Socket head Cap Screws (motor mounts)	M3/M4/M6 studding
M3 × 4, 10, 16, 20 Socket Head Cap Screws	Pop rivets
M4 × 10, 16, 20, 25 Socket Head Cap Screws	Adhesive-backed foam strip
M5 × 16, 20, 25 Socket Head Cap Screws	Adhesives
M6 × 16, 25, 40 Socket Head Cap Screws	Lubricants

There is a limited range of springs, spur and bevel gears, gear racks and bearings available – see the IDP website or see the demonstration board in EIETL. Specific structural materials, springs, spur and bevel gears, gear racks and bearings etc must be booked out using a materials request and given to the Mechanical Demonstrator at Design Acceptance. Fasteners, lubricants are available in the workshop. Adhesives are available in the EIETL, please only use in one the sheets provided.

K.1.2 Electrical Components

Standard Team Kit

	No. /team		No. /team
Line Following PCB	1	Prototyping PCB	1
Infra red detector amplifier assembly	1	Infra red emitter/detector assembly	4
Thermistor assembly & thermocouple materials*	1	Infra red Sensor test board	1
5v power supply lead (prototyping use only)	1	Motor/gearbox to PCB header lead	1
Microcontroller to PCB 12v lead	1	Breadboard + starter kit	1

Note that components will be issued by the Project demonstrators. Other than the components listed above, no reasonable limit is placed on the numbers of each component you may use. A record will be kept of materials used by each team and demonstrators may wish to check your circuit before issuing components.

Available 'Electronic' Components

Microswitch	Integrated circuit sockets
PCB headers and Connectors: (2, 3 &5 way)	Crimp terminal
Push button switch, SPST, PCB mounting	Relay, 12v DPCO, PCB mounting

Resistors:

CFR carbon film	0.25 W 5%

Available in decade ranges of 10^0 to 10^5 in the following values: 10, 12, 15, 18, 22, 27, 33, 39, 47, 56, 68 and 82 (up to a maximum of 10^6 ohms)

Potentiometers:

10 KΩ &500Ω multiturn (PCB)	10, 22 & 200Ω single turn (PCB)
100KΩ single turn (panel)	

Capacitors:

10nF 50v multilayer ceramic	0.1μF 50v multilayer ceramic
0.22μF 50v multilayer ceramic	0.47μF 50v multilayer ceramic
1μF 50v multilayer ceramic	0.1μF 35v tantalum (long lead positive)
0.22μF 35v tantalum (long lead positive)	4.7μF 16v tantalum (long lead positive)
10 μF 16v tantalum (long lead positive)	22 μF 16v tantalum (long lead positive)
47 μF 16v tantalum (long lead positive)	470 μF 16v Electrolytic

Range of 74C series IC's :

74HC00	Quad 2 i/p NAND gate	74HC08	Quad 2 i/p AND gate
74HC14	HEX Scmitt-trigger inverter	74HC74	Dual D-type flip flop
74HC85	4-bit comparator	74HC112	Dual J-K flip-flop
74HC123	Dual retriggerable monostable	74HC273	8-bit D-type flip flop
74HC393	Dual 4-bit binary counter		

Other active devices

LM358	Dual OP Amp	ICM555	Timer
LM311	Comparator	ZVN4306A	Enhancement MOSFET driver
PCF8574	8 bit expander for I^2C-Bus	3mm LED	Red/Green/Orange/Blue
	Reed Switch	Diode	1N4001
Diode bridge	W005	7805	5v Regulator
L293	Push/Pull 4 channel driver	LTC 1152	'Zero drift' op amp*
SFH 313FA	Phototransistor*	GP2Y0A21YK	Distance sensor*
SS495A	Hall Effect Sensor*	Strain Gauges beam	Strain gauges mounted on
5mm LED	White	IDPPic	12F1840 Pic module

* if required by the task set

Note: A handout giving general information on HCMOS logic and data sheets on relevant devices is available from the IDP website.

K.1.3 Software Components

Software Library	1
Microcontroller module	1
Power supply unit plus output lead	1
Al Chassis kit	1
Sensor simulation pcb + I^2C cable	1