A Trilobot interface in MATLAB/Simulink

H.L.Hagenaars
Report No. DCT 2004.64

Report of internal traineeship

Coach: Dr. Ir. P.Lambrechts (TU/e)
Supervisor: Prof. dr. H. Nijmeijer (TU/e)
Abstract

Trilobot is a mobile robot that can be controlled using a wireless serial link. A previous study made clear that communication with Trilobot running in its standard 'command mode' is very ineffective and therefore very slow. Furthermore this approach requires a lot of programming. In this study, several approaches will be investigated that can speed up the serial communication with Trilobot. Also an interface in Simulink will be developed to enable users to experiment with Trilobot without any programming knowledge.
# Contents

1 Introduction ................................................................. 1
   1.1 Trilobot as experimental setup ........................................... 1
   1.2 Previous research on Trilobot ............................................. 3
   1.3 Problem formulation ...................................................... 4

2 A Trilobot Interface in Simulink using serial communication .......... 5
   2.1 Serial communication in Windows ........................................ 5
   2.2 Integrating external code in Simulink ................................... 7

3 Trilobot in Command Mode ................................................ 9
   3.1 Command Mode programming ................................................ 9
   3.2 Importing sensor data from Trilobot ..................................... 10
   3.3 Conclusions ............................................................... 11

4 The 80C32 microcontroller ................................................. 13
   4.1 General operation .......................................................... 13
      4.1.1 Memory .............................................................. 13
      4.1.2 Special Function Registers ......................................... 15
   4.2 On-chip features .......................................................... 15
      4.2.1 Timers .............................................................. 16
      4.2.2 Integrated UART .................................................... 16
      4.2.3 Interrupts .......................................................... 16
   4.3 The Trilobot microcontroller ............................................. 17
   4.4 Running user programs on Trilobot ..................................... 18
Chapter 1

Introduction

Conducting experiments to test control strategies is a widely used approach in the field of control engineering. Experimental setups that are available to the control engineer are often ready to use, especially in education. The operation and user friendliness of experimental setups however, is not self-evident. Preparing an experimental setup and make it suitable to conduct experiments, requires in depth knowledge of the hardware of the setup. Furthermore, software drivers have to be written to be able to operate the setup, which requires knowledge of software engineering. Finally the engineer who prepares the experimental setup must fit the needs of the control engineer that will use the setup to experiment with.

In this report, the preparation of an experimental setup to make it suitable to experiment with, will be discussed. The setup considered in this report has some intelligence built in, by means of an 8-bit Central Processing Unit (CPU), which makes this setup an embedded system [7]. Therefore, the problem tackled in this report also involves a study of the basics of embedded systems programming.

The experimental setup used in this research is the Trilobot mobile Robot from Arrick Robotics shown in Fig. 1.1 [2]. In this chapter the features of the Trilobot will be discussed. Furthermore, previous research done on Trilobot will be discussed. At the end of this chapter, the research goals for this report will be formulated.

1.1 Trilobot as experimental setup

Trilobot is a mobile robot that consists of a base with two motor driven wheels (drivewheels) and a castor. There is an optical encoder attached to each driven wheel. The Trilobot is normally powered by a battery pack but in the setup used for this report, the battery pack is replaced by a rechargeable battery. A controllable gripper is mounted between the two drivewheels. Furthermore, the base is equipped with eight independently readable whiskers and a water sensor.

The top of Trilobot is called the mast and there are several sensors and controllable features mounted on the mast, including:

- Electronic Compass;
1. Introduction

- 4 light level sensors;
- Temperature sensor;
- Tilt sensors;
- IR communication transmitter;
- Green and red LED.

In front of the mast, a controllable head is mounted, driven by three servos. The features mounted on this head are:

- IR communication receiver;
- Sonar range finder;
- Passive infrared motion detector;
- Headlight;
- Sound detection;
- Laser pointer.

Figure 1.1: The Trilobot mobile robot

All the features of Trilobot are controlled via a controller board with a main microcontroller that holds Trilobot’s software, and two microcontrollers used as intelligent co-processors. There is a display and a keypad available to operate the software.
1. Introduction

The software that runs on Trilobot’s controller has several operating modes. The joystick control mode can be used to control the robot using the joystick. The Trilobot comes with a joystick that can be attached to the joystick connector mounted on Trilobot. The joystick can be used to control the drivewheels, the head and the gripper functions. The IR control mode can be used to control the robot with the IR remote control that comes with Trilobot. Many of Trilobot’s functions can be controlled using the remote control when Trilobot is in IR mode. For the serial communication, the Console Command Mode can be used. Making use of a high level programming language, simple ASCII commands can be sent to Trilobot from a desktop PC using the wireless serial link or using a cable. Responses can be received using the same serial link. There is a wide range of commands available that can be used to control the drivewheels, gripper and head and to request data of all the sensors on board Trilobot. This is a flexible method because the serial port can be controlled using a high level programming language such as C, C++, BASIC or Pascal.

Another, very powerful function of Trilobot is to run programs written by the user that can be uploaded to Trilobot and run directly from the available memory of Trilobot’s processor. Using a user program, Trilobot can operate autonomously without any communication with an external PC. Furthermore a user-defines mode can be written to communicate with an external PC.

1.2 Previous research on Trilobot

Previous research mainly focussed on the operation and functionality of Trilobot [8]. To test the suitability of Trilobot as an experimental setup, a simple tracking controller was implemented using the RS-232 interface and Q-BASIC programming with Trilobot in console command mode. To obtain information on the global position of the robot, use had been made of the encoder data in combination with the compass data. The performance of the implemented tracking controller was rather poor. The main reasons for the bad tracking behavior were:

- Using command mode programming, the smallest possible steering input is three degrees and the accuracy of the compass is two degrees. This lack of resolution leads to inaccurate tracking.
- The loop time of the program is relatively high, about 0.8 seconds.
- It is not possible to perform operations on Trilobot in parallel. A new operation can only be performed after the previous operation is done.
- The time to retrieve compass data is approximately 0.5 seconds.

Recommendations to improve the performance of the tracking controller and make Trilobot more attractive as an experimental setup included:

- Investigate the IR communication with Trilobot to avoid using the RS-232 interface.
- Try to speed up the communication between Trilobot and PC by making use of another programming language and/or avoiding command mode programming.
1. Introduction

- Make an interface in MATLAB/Simulink, that makes it easy to experiment with Trilobot for future users.

1.3 Problem formulation

The recommendations given in the previous research project are very diverse. The main problem using the Trilobot seems to be the serial communication in combination with the limited options offered by the command mode. When one wants to get sensor data, a command has to be sent to Trilobot. Trilobot will send back the requested data. The commands to request data have a minimum size of four bytes. The replied data are two bytes or four bytes. Suppose, for example, three sensors have to be read that return 4-byte values, it takes a transfer of 12 bytes only to request them and another 12 bytes to read them. If Trilobot could send a string with the data of the three sensors at a fixed rate, it would only take a transfer of 12 bytes to read those sensor data. This would cut the loop time in half and communication via the serial port may no longer be the factor that limits the loop time.

Furthermore, implementing a simple tracking controller requires considerable programming knowledge. It would be ideal if future users can use an interface in MATLAB/Simulink and use the realtime workshop to experiment with Trilobot without having to program a single character.

All these considerations lead to the following problem formulation:

*Improve the usefulness of Trilobot as an experimental setup with respect to the obtainable sample rate and the user interface.*

To solve this problem the following research goals have been stated:

- Make an interface in MATLAB/Simulink that makes communication with Trilobot via the RS-232 interface possible without any programming.

- Investigate if and how a reasonable sampling rate can be obtained using the RS-232 interface.

How to set up an interface in MATLAB/Simulink will be discussed first and an interface will be developed that can be used to read out sensor data when Trilobot is running in Command Mode. The performance of this approach will be tested with experiments. To write effective user programs for Trilobot, there has to be a basic understanding of Trilobot’s microcontroller. How Trilobot’s microcontroller works and how to program it will be covered next.

Subsequently, an interface that can communicate in both ways with Trilobot (send and receive) while running a user program will be developed, as well as a user program template. The performance of this approach will be tested.

Finally some other possible approaches will be discussed and this report will end with drawing conclusions and giving recommendations for future research.
Chapter 2

A Trilobot Interface in Simulink using serial communication

To make Trilobot attractive as an experimental setup for future users, a solution has to be found that enables users to control Trilobot using an external desktop PC with a serial link to Trilobot, without any programming. MATLAB/Simulink can be used to test control strategies and algorithms without any knowledge of programming. If there is an interface-block available in Simulink that takes care of the serial communication with Trilobot (i.e. import the sensor data in Simulink and send data and requests to Trilobot) the user can easily test control strategies and algorithms using the Real-Time Workshop. As a first step, a Simulink interface will be developed to receive data from Trilobot running in Command Mode.

In this chapter, first accessing the serial port in Windows2000 using C++ will be discussed to write routines to control Trilobot using a serial link. How to call these routines from within Simulink will be explained in section 2.2, which will result in a Trilobot Interface in Simulink. A detailed explanation of serial communication using the RS-232 protocol can be found in Appendix A.

2.1 Serial communication in Windows

Trilobot's Command Mode can be used to communicate with an external PC. Commands that define a certain operation can be sent to Trilobot via the serial port. After receiving a command, the software will perform the requested operation. Such an operation can be either a request for sensor data which will be returned to the PC via the serial port, or an operation to control one of the other features. The desktop PC must run software routines that can take care of this communication.

If the routines needed to setup communication with Trilobot are written and compiled as a Dynamic Link Library (DLL), those routines are accessible for other programs. When writing the code for the Simulink interface blocks, these routines can also be called.

DLL's are an important aspect of Windows. A DLL is a library of functions that a program can link with dynamically.

Dynamic links are only created when needed (as opposed to a static link that links in all the
2. A Trilobot Interface in Simulink using serial communication

Table 2.1: Function of the routines defined in command.dll

<table>
<thead>
<tr>
<th>routine</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CmdTRILOOpen</td>
<td>Open and configure the serial port</td>
</tr>
<tr>
<td>CmdTRILOStart</td>
<td>Start the communication with Trilobot</td>
</tr>
<tr>
<td>CmdTRILOStatus</td>
<td>Check whether the communication has started</td>
</tr>
<tr>
<td>CmdTRILOTerm</td>
<td>Terminate the communication with Trilobot</td>
</tr>
<tr>
<td>CmdTRILOOutput</td>
<td>Returns an integer with the value of the requested sensor</td>
</tr>
</tbody>
</table>

code of a library file). When a program needs a function that is not in the executable file, Windows loads the dynamic link library (the DLL), making all of its functions available to the application.

In this report use has been made of the C++ programming language to write a DLL with routines that can be used to communicate with Trilobot, but many other languages are suitable for this purpose. How to write a DLL using the C++ programming language will not be covered in this report (see [6]). The software discussed in this report is written for the Windows2000 operating system and may need to be adapted to work on other platforms.

To access the serial port under Windows using C++, use can be made of various Windows API (application programming interface) functions. Windows API functions are also defined in DLL's included in Windows. Which API functions are available for serial communications and how to use them will not be explained in detail in this report. For more information on these topics see [6].

Although serial communication is pretty much the same as normal file input/output, there arise some difficulties when dealing with serial communication. Those difficulties provide a limitation to what can be achieved using serial communication.

- Windows will place the received data in a receive buffer. Data can be read from this receive buffer and used for further processing. If the buffer is filled faster than data can be processed, the data to be read are not up to date. This will introduce a time-delay. It is important that the buffers do not get full and overflow.
- It is important that the receive buffer is empty on startup. If one tries to read 4 bytes in a row, but on startup there is already one byte present in the buffer, the data will be garbled.
- The data sent out by Trilobot are hexadecimal bytes or words sent as 2 or 4 ASCII characters. After reception, these data have to be converted to floating point numbers to be used in Simulink.

Appendix D.1 gives a source listing of the DLL file that contains routines setup and stop the serial communication with Trilobot, and to request data from the temperature sensor from Trilobot running in command mode. The routines that are defined in the DLL file are given in table 2.1.

The commands all begin with Cmd, which indicates that these routines are for use in combination with Trilobot's Command Mode. To setup, run and stop the communication, those
2. A Trilobot Interface in Simulink using serial communication

routines can be used as follows:

1. Calling `CmdTRIL0Open` will set the serial port and clear the buffers.

2. When calling `CmdTRIL0Start`, the communication will be started. The software will send a request to Trilobot and after receiving the requested data, the data will be converted from ASCII characters representing a hexadecimal number, to a floating point number. The routine `CmdTRIL0Status` can be called to check whether the communication actually started (i.e. if a request for data has been sent to Trilobot).

3. When the serial communication is running, the function `CmdTRIL0Output` will return the real time value of the requested sensor.

4. To terminate the communication, the function `CmdTRIL0Term` can be called.

2.2 Integrating external code in Simulink

To use the routines defined in a DLL within Simulink use can be made of an S-function block. An S-function is a computer language description of a Simulink block. S-functions can be written in MATLAB, C, C++ or various other languages. For the purpose pursued in this report, the S-function will be written in C which will be compiled as a C MEX-file. A MEX file is a library file executable from within MATLAB compiled using the `mex` command in MATLAB. On a windows platform, the extension of the MEX-file will be *.dll.

A C MEX-file that defines an S-Function block must provide information about the model to Simulink during the simulation. As the simulation proceeds, Simulink, the ODE solver, and the MEX-file interact to perform specific tasks. These tasks include defining initial conditions and block characteristics, and computing derivatives, discrete states, and outputs [3]. There is a template available for writing C MEX-files.

The source of the S-function used to communicate with Trilobot is listed in Appendix E.1. Wintarget can be used to translate the Simulink model into a Real-Time Application (RTA) for Windows. Wintarget is a real-time target for MATLAB Real-Time Workshop running under Windows. When building the RTA, the DLL containing the routines listed in table 2.1 has to be copied into the Wintarget directory. In the Simulink interface, a batch file is available that will take care of that. Appendix ?? gives a summation of the steps to take to setup the Trilobot interface.

In the simulation parameters of the Simulink model, the sampling frequency can be set. Note that, although the sampling frequency can be chosen freely, there is a limit to the sample frequency that can be obtained in requesting data from Trilobot. If the data refresh only once every second and the sample frequency set in the simulation parameters is 10 Hz, Simulink returns ten times the same value during one second according to the zero order hold principle. In order to investigate the real update rate of the data in the following chapters, the time between the reception of data will be measured in the routine that performs the serial communication.
2. A Trilobot Interface in Simulink using serial communication
Chapter 3

Trilobot in Command Mode

An interface has been developed that can be used to import sensor data from Trilobot into Simulink. The easiest way to obtain sensor data and to control Trilobot’s functions is in the Trilobot command mode. In this chapter, importing several sensor data in Simulink using the developed interface while Trilobot is running in command mode will be discussed and finally conclusions will be drawn.

3.1 Command Mode programming

In Command Mode, all of Trilobot’s functions can be accessed by sending commands to Trilobot. Each Command starts with an exclamation mark (!) followed by the ID number of Trilobot’s controller. This ID number can be set manually in the software. By default this ID number is one. However, when more than one Trilobots are used using the same serial data stream, the ID number can be changed so that each Trilobot only acts on the signals that contain its unique controller ID. [2]

The actual command starts with a character identifying the type of command. Trilobot distinguishes three types of commands. Put commands, used to send information to Trilobot are indicated with a "P". Get commands, indicated with a "G", can be used to retrieve information from Trilobot. Finally there are some other commands indicated with an "O". The character that indicates the type of command is followed by some parameter. For example, the command !PL03 will turn on the headlight and the command !GW1 returns a hexadecimal byte (8 bits) of which each bit represents the status of a whisker (a zero means inactive, a one means active). A complete list of available commands can be found in [2]. To request sensor data only the 'Get' commands have to be used.

As mentioned before, it is very important to use the same settings for both serial ports. The serial port settings of Trilobot’s serial port in command mode are a baud rate of 9600 baud, 8 data bits, no parity and one stop bit. In this chapter, retrieving the data of the compass, the sonar and the temperature sensor will be used to test the performance of the developed interface while Trilobot runs in command mode. The commands to retrieve those sensor data are given in table 3.1.
3. Trilobot in Command Mode

Table 3.1: Commands used to obtain sensor data

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Command</th>
<th>Returning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compass</td>
<td>!1GCl</td>
<td>Heading in degrees (0-360) as hex word</td>
</tr>
<tr>
<td>Sonar</td>
<td>!1GS1</td>
<td>Sonar distance in inches as hex byte</td>
</tr>
<tr>
<td>Temperature</td>
<td>!1GF1</td>
<td>Temperature in degrees F as hex byte</td>
</tr>
</tbody>
</table>

3.2 Importing sensor data from Trilobot

The results for importing the compass data in Simulink with the developed interface can be found in Fig. 3.1. The top figure shows the compass data for about 50 seconds. When rotating the Trilobot randomly, the compass heading varies between 0 and 360 degrees. The bottom figure shows a histogram of the elapsed time between two subsequent receptions of data. As can be seen in the figure, this time has an average of about 330 milliseconds. This means that a sample frequency of only 3 Hz can be obtained when retrieving the compass data only. Note that these data are captured by the routine that takes care of the communication with Trilobot not within Simulink.

![Figure 3.1: Compass data retrieved in command mode](image)

Fig. 3.2 shows the results obtained when retrieving only the data of the sonar for about 40 seconds. The elapsed time between the reception two of data is considerably smaller than is the case when receiving compass data. The average time required to retrieve sonar data is about 90 milliseconds. This means that a sampling frequency of 11Hz can be obtained when retrieving sonar data. Reading out the data of the temperature sensor (see Fig. 3.3) takes the same amount of time as reading out the sonar, about 90 milliseconds.

When one wants to obtain the data of the three sensors continuously via the same serial data stream, the same operations have to be performed in a sequential way. This implies that it takes more than half a second to retrieve the data from the three sensors, which leads to a
3. Trilobot in Command Mode

![Graph 1](image1.png)

Figure 3.2: Sonar data retrieved in command mode

![Graph 2](image2.png)

Figure 3.3: Temperature retrieved in command mode

3.3 Conclusions

When looking at the results, the first thing that stands out is that reading out the compass takes much more time than reading out the other two sensors that are considered. One part of the explanation for this, is the fact that the compass heading is sent to the computer as four ASCII characters representing a hexadecimal word. The other two sensor data arrive at the serial port of the PC as two ASCII characters representing a hexadecimal byte. The baudrate of 9600 baud however, makes a much higher transfer rate possible than
the 3Hz obtained with the compass.
The real explanation of the low sampling frequency that is obtained with the compass can be found in the compass data sheet. The compass is a vector 2D electro magnetic compass. The data sheet specifies a maximum output rate of five times per second. So the maximum sampling frequency that can be obtained when the serial communication is not taken into account, is 5Hz. So when one wants to use the compass data in any kind of way, the maximum achievable sampling frequency will be 5 Hz.

Reading out the sonar data and the temperature sensor is much faster and the time required to retrieve those data may suffer from the serial communication. If one wants to obtain the data at a constant rate, it is not necessary to request the data each time and then wait for Trilobot to sent the requested data back. It would be much better to have a program running on Trilobot that keeps sending the data of one or more sensors to the desktop pc at a constant rate. The software running on the pc only has to read those data and convert them to floating point numbers. This could cut the number of bytes to be transferred in half.

Unfortunately there is no such program available in Trilobot’s software. However, Trilobot can also run programs written by the user. Files, in Intel hexadecimal format, can be downloaded into Trilobot’s free memory space. There is a large number of predefined functions and variables available in Trilobot’s software. A jump table defined in the file trilodef.asm, that can be downloaded from the Trilobot website, provides access to those predefined functions and variables. Writing programs for Trilobot’s processor however, requires specific knowledge of the microprocessor used to control Trilobot. Any high level language can be used (e.g. C or PASCAL) to write those programs, as long as they are compiled as Intel hex files. Because of the small program space and the predefined I/O map (that provides routines to be used in assembly language), it is recommended to write the programs in assembly language.

The next chapter will explain the basics of the features and operation of the microprocessor on board Trilobot. Furthermore, the use of assembly language to write programs for Trilobot’s microcontroller efficiently and how to access the specific features of that microcontroller, will be addressed.
Chapter 4

The 80C32 microcontroller

The main CPU of Trilobot is an Intel 80C32 Microcontroller. The 80C32 microcontroller is part of an entire series of microcontrollers, the 8051 series, initially designed by Intel in the 1980's. The microcontrollers that are part of this family all use the 8051 core, licensed from Intel. The core refers to the instruction set and special function registers that can be used to access the on-chip peripherals including timers, counters and UART's. Many other manufacturers produce 8051 compatible microcontrollers (e.g. Philips, Atmel and Dallas Semiconductors). In this chapter, the general operation and features of the 8051 will be explained. In the last section the specific configuration of Trilobot's microcontroller will be discussed. The basics of assembly language programming for the 80C32 microcontroller can be found in Appendix B

4.1 General operation

The 8051 microprocessor has an instruction set that defines 255 instructions. Instructions are operations that can be carried out by the processor, such as addressing memory and performing mathematic operations. The 8051 operates based on an external crystal. This crystal is an electronic device that emits pulses at a fixed frequency when power is applied. The crystal frequency of the 8051 microcontroller is 11.059 MHz. The 8051 uses the crystal to synchronize its operations. Effectively the 8051 uses so called 'machine cycles'. A single machine cycle is the minimum amount of time in which a single 8051 instruction can be executed. Many instructions however take multiple cycles. In the 8051, a machine cycle takes 12 crystal pulses. The memory of the 8051 can be accessed by 8-bit addresses which can be quickly stored and manipulated by the 8-bit CPU.

4.1.1 Memory

The 8051 has three types of memory, i.e. on-chip memory, external RAM (Random Access Memory) and external code memory. The on-chip memory consists of 128 bytes of internal RAM and 128 bytes of special function registers (SFR's). To effectively program the 8051 it
is necessary to have a basic understanding of these memory types. Fig. 4.1 gives a schematic representation of the memory configuration of the 8051.

Internal RAM
The 8051 has 128 bytes of internal RAM that can be found on-chip so it is the fastest RAM available and it is very flexible in terms of reading, writing and modifying its content. Internal RAM is volatile, so it will be cleared when the microcontroller is reset. A schematic representation of internal RAM is given in Fig. 4.2. The first 32 bytes (00h-1Fh) are reserved for four register banks each of 8 bytes. The next 16 bytes (20h-2Fh) are reserved as bit memory containing 128 bits. The 80 bytes of internal RAM remaining (30h-7Fh) can be used to store data that have to be accessed frequently or at high speed. This area is also utilized by the microcontroller to store the operating stack.

Register banks
The 8051 uses eight 8-bit R registers that are used in many of the instructions that can be performed, e.g. manipulating values and moving data. These R registers are numbered from 0 trough 7 and together they form a register bank. There are four of these register banks. On startup of the processor, register bank 0 (00h-07h) is used by default. One can instruct the processor to use another register bank. The register banks that are not in use can be used as extra internal RAM. It is important to know which register bank the microcontroller
uses. If variables are stored in the addresses of the register bank in use, those variables will be overwritten while the processor carries out instructions.

**Bit memory**

Bit memory provides 128 bits that can be accessed separately using special instructions (see Appendix B). These 128 bits can also be used as another 16 bytes of memory.

**External code memory**

On the 8051 microcontroller, the external code memory is provided in the form of EPROM (erasable programmable read only memory) which implies that this memory is read only. This memory is limited to 64K. Code space is mostly used to hold the actual 8051 program that is to be run.

**External RAM**

As opposed to Internal RAM, the 8051 also supports external RAM. External RAM is read/write. Accessing external RAM costs more machine cycles than accessing internal RAM. This makes external RAM slower. The capacity however is much larger than the internal RAM. While internal RAM is limited to 128 bytes, the 8051 supports external RAM up to 64K.

### 4.1.2 Special Function Registers

128 bytes of the on-chip memory of the 8051 microcontroller are used as Special Function Registers (SFR's). These SFR's are areas of memory that control specific functionality of the 8051 processor. In this section, the SFR's used in the user program listed in appendix F are discussed. A complete listing of the SFR's and how to use them can be found in [5].

**The Accumulator ACC**

The accumulator register is used as a general register to accumulate the results of a large number of instructions. It can hold a 1-byte (8-bit) value and is the most versatile register of the 8051 because of the large number of instructions that make use of it. More than half of the 8051's 255 instructions manipulate or use the accumulator in some way. If one wants to add the values 10 and 20, the resulting 30 will be stored in the accumulator. After the operation, the value in the accumulator can be processed. The on-chip RAM address of the accumulator is E0h. So instead of using ACC to access the accumulator, one can use E0h also. The accumulator can also be used to store an 8-bit variable temporarily. Note however, that this value may be overwritten when carrying out other instructions.

**The Data Pointer (DPTR)**

The Data Pointer is the only user accessible 2-byte (16 bit) register in the 8051. DPTR is used to point to data. It is used in addressing external memory. It can also be used to temporarily store 16-bit variables.

### 4.2 On-chip features

The 8051 microcontroller has several integrated features that can easily be accessed by the programmer. The three most relevant features will be discussed in this section.
4.2.1 Timers

To keep track of time in a time-critical application, the 8051 has two timers that can be used to time events with high precision. Those two timers can be controlled, set, read and configured individually by specific special function registers, which will not be discussed in this report [5, 4]. The timers have three general functions:

1. keeping time and calculate amount of time between events;
2. counting events;
3. generating baud rates for the serial port.

To keep time and count time between events, each timer can be configured as an interval timer. An interval timer counts upwards and is incremented by one every machine cycle. As mentioned before a machine cycle consists of 12 crystal pulses, so the timer will be incremented $\frac{11059000}{12} = 921583$ times per second. This information can be used to time events and to set a fixed loop frequency in a program. The timers can also be used as an event counter, to count certain events. Furthermore the timers can be used to set the serial port baud rate. Which special function registers can be used to read out and configure the two timers can be found in [5, 4].

4.2.2 Integrated UART

One of the powerful features of the 8051 microcontroller is the integrated UART. This integrated UART makes it easy to read and write values to the serial port without accessing all the pins of the serial port separately to clock out each individual bit, including start bits, stop bits and parity bits. The internal UART can be configured simply using the special function registers and reading from and writing to the UART comes down to reading from and writing to a specific special function register. The baud rate can be either based on the crystal frequency (which will result in 921583 baud) or it can be specified using one of the timers. How to configure the serial port, how to set the baud rate and how to read from and write to the serial port can be found in [5, 4].

4.2.3 Interrupts

An interrupt is a special feature that allows the 8051 to provide the illusion of multitasking. An interrupt is an event that interrupts the normal program execution. If an interrupt occurs, the normal program flow will be put on hold and a special interrupt routine will be executed. When the interrupt routine is executed the program will resume the normal program flow as if nothing happened. The 8051 microcontroller can generate an interrupt on the following events:

- Timer overflow;
- Receiving a character via the serial port;
- Transmitting a character via the serial port;
4. The 80C32 microcontroller

- Two external events.

If a special routine has to be executed when a character arrives at the serial port, one can check the serial port once every loop. But if there are no characters available at the serial port, this takes unnecessary machine cycles. If serial interrupts are activated, one does not have to check the serial port once every loop, but the microcontroller does it, and it jumps to a specified interrupt routine that has to be stored at a specified interrupt handler address. An interrupt routine is the same as a normal sub-routine but it has to end with the RETI (return from interrupt routine) instruction in stead of the RET instruction (see Appendix B). The interrupts can also be controlled using a SFR. The address to which the program jumps when an interrupt occurs is fixed for each event. When the serial interrupt is configured for example, the program jumps to address 0023h when a character is transmitted or received. How to configure the interrupts can be found in [5, 4].

4.3 The Trilobot microcontroller

Fig. 4.3 gives a schematic representation of the external memory of the 80C32 microcontroller that is used to control Trilobot. The top 32K of program space (EPROM) holds the system software. The top 32K of Data space (external RAM) holds the predefined hardware I/O map that can be used to control the sensors and servos of Trilobot. There are also several routines available for serial communication, IR communication etc. These predefined routines can be controlled through the functions defined in the trilodef.asm that can be downloaded from the Trilobot website. The predefined subroutines can be called with the LCALL instruction and the routines end with an RET instruction that causes the program to jump back to the main program loop. When requesting certain data, those data will be loaded into the DPTR register or the ACC register depending on the size of the data (16-bit or 8-bit).

The lower 32K of code space contain data for speech and interrupt vectors that are used to configure interrupts. Programs can be uploaded to the lower 32K of free data space. When selecting 'RUN FROM RAM' from the 'RUN USER PROGRAM' menu, the lower 32K of data space will be copied into EPROM and the program will be executed at the expense of speech.

When the lower 32K of data space is copied into the lower 32K of code space,
the interrupt vectors are copied to data space. It is important to prevent the program from writing to the upper 32K of code space, because the system software resides there.

Some of Trilobot's devices are not directly connected to the main microcontroller. There are two other microcontrollers in use as intelligent coprocessors. The coprocessors perform motor control and other time consuming tasks. Fig. 4.4 shows a block diagram of the electronics on board Trilobot.

![Block diagram of the electronics](image)

**Figure 4.4: Functional block diagram of the electronics**

### 4.4 Running user programs on Trilobot

User programs can be downloaded into Trilobot only in Intel hex format. Intel hex format can be created using an assembler or compiler and contains addresses and data in text format that can be viewed with a normal text editor. The assembler used to generate Intel hex files from the user programs written in assembly language that are discussed in this report, is an assembler from Raisonance [10] that is integrated in the Raisonance Integrated Development Environment (RIDE). This development environment can be used to write, assemble and test applications for a wide range of microprocessors, including the 8051 family. All the programs start above 0100h to avoid the program from overwriting the interrupt vectors.

To download the Intel hex file into Trilobot the 'DOWNLOAD/UPLOAD' option has to be selected from Trilobot's menu. When the down arrow key is pressed, Trilobot waits for the download process to start. The assembled program can be send to the serial port at PC side using the DOS command `copy`. Note that the settings of the serial port have to be 9600 baud, no parity, 8 data bits and one stop bit. These settings can be adjusted in DOS with the `mode` command. In the user program interface in Simulink (see section 5.4), a batch-file can be used to configure the port and start the download.

After a successful download, the program can be run from RAM by entering the starting address.
Chapter 5

User programs on Trilobot

To speed up the communication with Trilobot, a user program can be written that only performs the desired operations and no other operations that cost valuable loop-time. Because Trilobot is a closed system (i.e. there is no information available to access the hardware directly), the assembly language file trilodef.asm defines a hardware I/O map that contains routines that can be called to access Trilobot’s hardware. In this chapter, the performance of the functions in the predefined hardware I/O map will be investigated. Furthermore the benefit of a user program over command mode programming will be made clear. After that, the problem of sending commands to Trilobot will be discussed which leads to a scalable example user program and a Trilobot interface in Simulink that can be used to communicate with Trilobot while running the user program. At the end of this chapter some other possible approaches to experiment with Trilobot will be discussed and conclusions will be drawn concerning the developed user program and interface.

5.1 Using the predefined hardware I/O map

The predefined hardware I/O map provides numerous functions to access Trilobot’s hardware. There are several commands to request sensor data which will be loaded either in the data pointer (DPTR) register or the accumulator (ACC) depending on the format of the data. To get an idea of how fast these routines are, the routines can be timed using one of the on-chip timers of Trilobot’s microcontroller.

<table>
<thead>
<tr>
<th>sensor</th>
<th>routine call</th>
<th># machine cycles [-]</th>
<th>corresponding time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoders</td>
<td>utlgod</td>
<td>10541</td>
<td>11.3</td>
</tr>
<tr>
<td>Sonar</td>
<td>songet</td>
<td>1992</td>
<td>2.2</td>
</tr>
<tr>
<td>PIR</td>
<td>utlgpir</td>
<td>22</td>
<td>0.0239</td>
</tr>
<tr>
<td>Compass</td>
<td>utlgc1</td>
<td>194900</td>
<td>211</td>
</tr>
<tr>
<td>Temperature</td>
<td>utlgct</td>
<td>1254</td>
<td>1.4</td>
</tr>
<tr>
<td>Light intensity</td>
<td>utlgll</td>
<td>16</td>
<td>0.0174</td>
</tr>
<tr>
<td>Whisker status</td>
<td>utlgw</td>
<td>6709</td>
<td>7.3</td>
</tr>
</tbody>
</table>
Table 5.1 gives the number of machine cycles needed to execute the sub-routine to load the data of the specific sensor into one of the registers. As explained in sub-section 4.2.1, the microcontroller can execute 921583 machine cycles per second, so one machine cycle takes approximately 0.0011 milliseconds. The last column gives the exact corresponding time it takes to read the specified sensor. There is a huge difference in the time it takes to read the sensors. This is due to the specifications of the sensor and the length of the sub-routine. It also takes some time to send the data to the serial port. There are some predefined routines available for serial communication, but they cannot be timed because both timers are in use by the serial routines.

Reading the compass takes about 21 milliseconds, which corresponds with the maximum sampling frequency of 5 Hz specified in the data sheet. As mentioned before, the use of compass data has to be avoided when one wants to obtain a sampling frequency above 5 Hz.

5.2 Importing sensor data from Trilobot

To compare the results of importing sensor data in Simulink in command mode and the results obtained with a user program, the same sensor data are imported in Simulink, i.e. sonar data and temperature. The compass data are omitted for the reason stated before. The user program simply requests the sensor data and writes them to the serial port, all using calls to the predefined I/O map.

Fig. 5.1 shows the results for retrieving the data of the sonar. The upper figure shows the value returned by the sonar in inches for about 41 seconds. The lower figure gives a histogram of the elapsed time between the reception of sensor data.

![Figure 5.1: Sonar data retrieved while running user program](image)

It takes about 30 ms to read the sonar data using a user program. In command mode it takes approximately 90 ms. So using a user program, a sampling frequency of over 30Hz can be
obtained as opposed to the 11 Hz obtained in command mode. For retrieving temperature, shown in Fig. 5.2, the same results are obtained.

Figure 5.2: Temperature retrieved while running user program

Retrieving a combination of sensor data however, is still a sequential process. Fig. 5.3 shows the elapsed time between the reception of data when both the sonar data and the temperature are retrieved. The time required to read both sensors lies between 40 and 50 ms. This still gives a minimal sample frequency of 20 Hz.

Figure 5.3: Retrieving both sonar data and temperature when running a user program

Note that the user program can send the data to the serial port at a higher frequency than the obtained sample rates. The serial communication slows down the process considerably.
Avoiding use of the command mode however decreases the number of bytes that have to be transferred via the serial link which leads to more acceptable sampling rates.

5.3 Sending data to Trilobot

Importing sensor data in Simulink at a constant rate has been achieved. To control Trilobot however, data have to be sent to Trilobot also. The predefined hardware I/O map also defines routines to control the drive motors, the servomotors of the mast and the several lights and LED’s. This report will focus on drive motor control because it is the most interesting for implementing control strategies.

5.3.1 Practical implementation

While sending data from Trilobot to the desktop PC, the Trilobot sends data once every loop, and the receiving PC processes these data. It is desirable that those data are up to date, hence a sample frequency as high as possible is desirable.

When sending commands to Trilobot, it has to act on the reception of data. It is not necessary to send a command once every loop. To let Trilobot drive in a straight line, for example, a command has to be sent, only when Trilobot deviates from the desired trajectory. So it is not needed to wait for a steering command once every loop, which obviously costs valuable loop time. This is solved by checking the serial input line once every loop and only if a character is received, the program runs the steering routine.

5.3.2 Drive motor control using the predefined I/O map

Trilobot has two drive wheels with encoders that can also be controlled using the predefined hardware I/O map. The drive wheels however, are already controlled using low level controllers implemented in Trilobot’s software. Coprocessor 2 is used to perform the motor control. There are no commands available to control the two drivewheels separately. The Trilobot can be moved forward and backward and can rotate over a certain angle. Keeping track of Trilobot’s orientation is very difficult because the information of both encoders can not be retrieved separately. When requesting the encoder data (i.e. the drive motor distance), Trilobot sends back the sum of encoder counts of both encoders mounted on each of the drive wheels (see [2]). Normally the compass is used to determine the orientation of the Trilobot, but considering the time it takes to read the compass, this is not a serious option.

5.4 The user program interface

The input and output of data can be combined in a Simulink model. A Simulink interface is developed that can receive sensor data and send data to Trilobot. The whole interface consists of a DLL file and two S-functions, as explained in chapter 2, that can be used to communicate with Trilobot in combination with a user program. The user program and both the library file and the S-functions can be considered a template which can be easily adapted to fit specific needs.
5.4.1 The assembly language user program template

In appendix F, a code listing is given of the user program that can be used as a template to communicate with the interface in Simulink. The user program has been written to send one 1-byte sensor value and one 2-byte sensor value to the desktop and to act on one steering command. Fig. 5.4 shows the program flow of the user program. At startup the battery is checked and a message indicating the battery status is shown on the display. After that the serial communication device and the sonar are initialized. After the initialization, the program waits for a key press to start. Pressing the Y key, starts the program loop and pressing the 1 key runs the help routine. The help routine displays a short text on how to use the program. In the main program loop, the sensor data are sent to the serial port and the program checks if there are any data received. If there are any data received, the program jumps to the serial input routine and performs the defined operations. The program loop can be interrupted by pressing the Y button, which causes the program to pause, or by pressing the N button, which exits the program by jumping back to the main menu of Trilobot’s software. For safety, the first dip-switch on top of Trilobot can be activated. If this switch is activated, Trilobot pauses if any of the eight whiskers is active.

![Program flow of user program](image)

Figure 5.4: Program flow of user program.a51

5.4.2 The Simulink interface

The Simulink interface that can be used in combination with the user program, consist of two S-function blocks. The S-function s.trilo.out.c (see appendix E.3) imports the two sensor values in Simulink. The first output is the 2-byte sensor value. It is important that this is
Table 5.2: Function of the routines defined in trilo.dll

<table>
<thead>
<tr>
<th>routine</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRILOOpen</td>
<td>Open and configure the serial port</td>
</tr>
<tr>
<td>TRILOStart</td>
<td>Start the communication with Trilobot</td>
</tr>
<tr>
<td>TRILOStatus</td>
<td>Check whether the communication has started</td>
</tr>
<tr>
<td>TRILOTerm</td>
<td>Terminate the communication with Trilobot</td>
</tr>
<tr>
<td>TRILODMDist</td>
<td>Returns a double with the 2-byte value of the requested sensor</td>
</tr>
<tr>
<td>TRILOPir</td>
<td>Returns a double with the 1-byte value of the requested sensor</td>
</tr>
<tr>
<td>TRILOSend</td>
<td>Sends a datastring to Trilobot</td>
</tr>
</tbody>
</table>

The routine TRILODMDist returns the 2-byte value sent first to the serial port in the user program. The second output is the 1-byte sensor output. To send commands to Trilobot the S-function s.trilo.in.c, listed in appendix E.2 can be used.

The routines that are used in the two S-functions are defined in a DLL of which the source is listed in appendix D.2. The routines defined in this library file are explained in table 5.2
5. User programs on Trilobot

If one wants to send for example two 1-byte values, the TRIL0Thread routine of the DLL can easily be adapted. Note that this is not a function that is called directly in one of the S-functions. This routine starts when calling TRILOStart. The TRILOSend routine sends data to Trilobot. The data string to send has to be specified in the DLL file.

The Trilobot interface, shown in Fig. 5.5, has three control buttons. The first one can be used to copy the DLL file into the Wintarget directory when changes have been made. The second button can be used to download the user program into Trilobot. Double clicking this button will send the assembled user program to the serial port using the right settings. Normally the user program stays in memory when power is off. Most of the times however, the user program has to be downloaded after power-up. This is probably due to the attachment of the rechargeable accu. The last button is to plot the output of both sensors after an experiment. The sensor data are available for processing in the workspace after an experiment.

5.5 Avoiding the use of the predefined hardware I/O map

To effectively control Trilobot’s drivewheels, the use of the predefined hardware I/O map has to be avoided. There are several reasons:

- It would be desirable if the data of both the encoders could be obtained separately and if the drive motors of the wheels could be controlled separately. In that case, one can get round the low level controllers implemented in Trilobot’s software and the control loop can be closed in Simulink. The Trilobot however is a closed system, and on which pins of the processor the various control signals operate has to be investigated in order to control the hardware directly. An extra difficulty is that the drive motors and encoders are not directly connected to the main processor but to coprocessor 2.

- Trilobot’s software makes use of both timers available on the processor. When using the serial port’s communication routines of the hardware I/O map, the timers cannot be used. Normally one of the two timers has to be used to generate the serial port’s baud rate, but then the other one is free to use, for example to set a fixed sampling rate. So writing specific serial port communication routines would result in an extra timer that can be used.

- When using user-written serial port routines, use can be made of serial interrupts which speeds up the communication.

- The data of the sensors are sent to the serial port as ASCII characters. An 8-bit value is sent to the serial port as two 8-bit ASCII values representing one hexadecimal byte. So to receive one byte of data, two bytes have to be transferred. This could be solved also when using user written serial communication routines.

5.6 Avoiding communication with an external device

To obtain really high sample rates, one can avoid the serial communication or even avoid communication with any external device completely. One way to do so, is to run the desired
control strategy as a user program on Trilobot. Trilobot can then run autonomously. The main drawback of this approach however, is that the control strategy to be implemented has to be programmed in assembly language. This requires in depth knowledge of the microprocessor and assembly language.

One way to overcome this problem is to develop a realtime target for the 80C32 microcontroller. In that case use can be made of the code generator of the Real-Time Workshop in Simulink to generate code from a Simulink scheme, made by a user without any programming, that can be downloaded into Trilobot’s microprocessor. Developing such a realtime target for the 80C32 microcontroller however, seems to be a very difficult task. The 80C32 uses 8-bit integers and does not support floating point arithmetic by itself. Also it is very difficult to get code, that makes use of 32 bit integers and is very dependant on floating point arithmetic (as is usually generated by Simulink’s code generator), to work properly on an 8-bit processor. Furthermore, a drawback of code generated by Simulink’s code generator is that it is not as efficient concerning memory usage as programs written directly in assembly language. This is a disadvantage because of the restricted memory space available in the 80C32 microprocessor.

Finally, the most important drawback is that the experimental data have to be saved somewhere in Trilobots memory and uploaded to an external pc after an experiment to analyse and visualize the data. Because the memory of the microprocessor on board Trilobot is small, the quantity of data that can be saved and consequently the runtime of an experiment is restricted.

5.7 Conclusions

An interface in Simulink to control Trilobot has been developed that makes use of a user program running on Trilobot’s CPU. The performance of this approach concerning the reception of data is considerably better than is the case when Trilobot is running in command mode. A sampling frequency of 20Hz can easily be obtained.

Also sending data to Trilobot in such a way that the user program acts on the received data has been obtained, which leads to an interface in Simulink and a scalable template for a user program.

The use of the predefined hardware I/O map however leads to restrictions on the drive motor control and the use of other features of the microprocessor such as the timers. Writing specific functions to access the hardware and serial port directly would have the following benefits:

- When the encoders and drive motors can be accessed directly, routines can be written and included in the user program that make it possible to implement controllers in Simulink and the control loop can be closed outside Trilobot which gives much more flexibility in what can be done with Trilobot.

- When serial port communication routines can also be implemented in the user program, effective use can be made of the serial interrupts, which leads to faster programs. Furthermore there is a timer available in that case which can be used to generate a fixed sampling frequency.

- Sending data via the serial link is very inefficient using the predefined serial routines.
5. User programs on Trilobot

Bytes to be sent are transferred to a 2-byte format. After receiving those two bytes they will be transferred back to one byte.

- When using user written serial port communication routines, serial port interrupts can be used.

Accessing the serial port directly is not very difficult (see chapter 4), so writing user defined serial port communication routines is recommended. The main drawback however is, that it is not known on which pins of the microcontroller the various hardware devices operate. In case of the drive motors and encoders there is an extra difficulty because they are not directly attached to the main controller. This has to be investigated if one wants to use Trilobot as an experimental setup to test more sophisticated control approaches.
Chapter 6

Conclusions and recommendations

6.1 Conclusions

Trilobot is a very versatile experimental setup because of its numerous features and the several approaches to communicate with it. This versatility and the user friendliness however, also results in a system not very suitable for control engineering purposes, because of the restricted sample frequency that can be obtained when using the standard operating modes. Also a lot of programming is involved in using Trilobot, which does not make it easy to use. To make it more easy to control, an interface in MATLAB/Simulink was designed to import data from the sensors into Simulink while Trilobot runs in Command Mode. To efficiently develop the interface, the concept of serial communication in Windows has been investigated and the way to include external code in a Simulink model.

To test the performance of the interface, the elapsed time between the reception of two subsequent samples of data was used as an indicator. This elapsed time is about 90 ms for both the temperature sensor and the sonar, leading to a sampling frequency of about 11 Hz. Reading out the data from the compass takes considerably more time, about 330 ms. This is because of the specifications of the compass. Its maximum obtainable sampling frequency, regardless of the way of obtaining the data, is 5 Hz. When retrieving, for example, data of two sensors, the sample time simply adds up because of the serial, i.e. sequential communication. This results in a rather poor performance when the command mode is used.

The high sample time is due to the way communication with Trilobot in Command Mode works. Before receiving data, the data have to be requested by sending a command to Trilobot via the serial link. To avoid these extra bytes to be transferred, a user program can be written to optimize the import of sensor data into Simulink.

To be able to write programs for Trilobot, its 80C32 microcontroller was discussed. All the features and restrictions of the 80C32 and how to program it were addressed. A user program was written that sends the data of two sensors to the desktop PC and acts on the reception of data in an efficient way using the predefined hardware I/O map. Furthermore, an interface was developed that can be used to communicate with Trilobot running the user program.

The time elapsed between the reception two subsequent data points, decreased considerably
to about 30 ms per sensor. When obtaining data from two sensors, a sample frequency of 20 Hz can be obtained. Timing of the routines made clear that the serial port communication is the limiting factor in obtaining a high sample rate. A sample frequency of 20 Hz is, in general, acceptable to experiment with.

Sending data to Trilobot from within Simulink does not lead to any problems. The use of the function provided in the predefined I/O map restricts the functionality of Trilobot and makes it difficult to use Trilobot for control engineering purposes. Control of the drive motors is very limited when only the predefined hardware I/O map is used. Avoiding the use of this predefined map and writing specific routines to access the drive motors and the encoders may considerably improve Trilobot's functionality.

Another approach to use Trilobot as an experimental setup was investigated, the development of a realtime target for the 80C32 microprocessor. Because of the fact that the 80C32 has an 8-bit CPU and does not support floating point arithmetic however, the development of a realtime target for the 80C32 is a very difficult task. Furthermore, the limited amount of memory available restricts the quantity of data that can be stored and used for further processing after an experiment.

6.2 Recommendations

In this report, the development of an interface in Simulink has been discussed. Importing sensor data has been achieved at an acceptable sample rate. The control of, for example, the drive motors, is far from optimal. To improve the drive motor control the use of the predefined hardware I/O map has to be avoided. Furthermore, to optimize the data transfer between the PC and Trilobot it is better to avoid the use of the predefined map completely. Recommendations for further research are:

- Investigate on which processor pins the different control signals work and try to write routines to effectively control the driven wheels.

- Write serial communication routines for Trilobot that optimize the transfer of data between Trilobot and the PC. Currently a 1-byte value is transferred as two 1-byte ASCII characters representing a hexadecimal byte.

- When avoiding the use of the redefined hardware I/O map, use can be made of the timers of the microprocessor. One timer will be needed to set the baud rate of the serial port. The other timer however, can then be used to set a fixed loop time, which leads to a fixed output time of the sensor data.

- When avoiding the predefined hardware I/O map, use can be made of the configurable interrupts of the 80C32. This feature can be used to optimize the loop time of the user program.
Bibliography


Appendix A

Serial Communication

The protocol used mainly for serial communication is the RS-232 protocol. The serial port is used to convert bytes sent to the port to a binary stream, as well as to convert a binary stream back to bytes. The serial port contains an electronic chip called a Universal Asynchronous Receiver/Transmitter (UART) that actually does the conversion.

Serial communication uses different formats to transmit data. Both endpoints have to use the same setting, otherwise the data transmission will result in garbled data. There are four parameters to configure the serial communication device, the baud rate, the number of data bits, the number of stop bits and the optional parity bit. Fig. A.1 shows a schematic representation of the transfer of a character. Each transmitted character is packaged in a character frame that consists of a single start bit followed by the data bits, the optional parity bit and the stop bit or bits. RS-232 uses only two voltage stages, called mark and space. Mark is a negative voltage and space is a positive voltage.

**Baud rate**
The baud rate is the measure of how fast data are transmitted between the two devices. The baud rate is equal to the maximum number of bits (including start, parity and stop bits) that are transmitted per second. For example, when there is one start bit, eight data bits, no parity and two stop bits, the character frame has 11 bits. When the baud rate is set to 9600, 9600/11=872 characters can be sent per second. Note that this is the maximum transmission
rate. It may happen that the hardware on one of the two ends is not able to reach this rate. The duration to transfer a single bit, the bit time, is the reciprocal of the baud rate.

**Start bit**
The start bit is the beginning of each character frame. It is a transition from negative to positive voltage to indicate the beginning of a new character.

**Data bits**
Data bits are transmitted upside down and backwards. That is, inverted logic is used and the order of transition is from least significant bit to most significant bit. To interpret the data bits in the character frame, one must read from right to left. A negative voltage is a one and a positive voltage is a zero. The binary byte that is transferred in Fig. A.1 is 1101101 (109 decimal, which is the ASCII code for m).

**Parity bits**
An optional parity bit follows the data bits in the character frame. The parity bit (if present) follows also inverted logic. This bit provides a simple way of error checking. The parity can be set to even or odd. In Fig. A.1 the parity is set to odd. The transmitter will set the parity bit in such a way that the sum of active data bits (negative voltage) and the parity bit is odd. In Fig. A.1 there are five active bits among the data bits, this is an odd number, so the parity bit is inactive (high voltage). If the parity bit was active, the sum of active data bits and the parity bit was six, which is an even number. This indicates that something went wrong during transfer.

**Stop bits**
The last part of a character frame consists of one or two stop bits. These bits are always represented by a negative voltage. If no further characters are transmitted, the line stays in negative condition. The transmission of a new character frame begins with a start bit with positive voltage.
Appendix B

Assembly language programming

Every type of CPU understands its own machine language. Instructions in machine language are numbers stored as bytes in memory. Each instruction has its own numeric code called its operational code, or opcode. Machine language is very difficult to program directly and deciphering the meaning of numerical-coded instructions is tedious for humans. Loading, for example, the value of register R0 into the accumulator ACC is encoded using the following hex code:

E8

In assembly language, each instruction represents exactly one machine instruction. For example, moving the data of R0 into the accumulator ACC would be represented in assembly language as:

```
MOV ACC, R0 ; move the values stored in R0 to ACC
```

Here the meaning of the instruction is much clearer than the machine code. The word MOV is a mnemonic for the move instruction. The text after the semicolon is a comment and will be omitted by the assembler. An assembly instruction has the general form:

```
name: mnemonic operand(s); comment.
```

The name can be used to give the memory location of the specific instruction a name that is far more easy to recognize than a memory address, which makes the code much more easy to read. An assembler can be used to convert the assembly language program into machine language. Because every assembly language instruction represents a single machine instruction, it can be used to write efficient programs and to keep the program as small as possible. This is a great advantage when the memory space of the microcontroller is limited as is the case with the 8051. When compiling a higher level language such as C, a single statement produces many machine language instructions which increases the size of a program and decreases the speed.
B. Assembly language programming

B.1 Addressing modes

There are several ways to address a given memory location. At first, the memory location can be expressed as a decimal, a hexadecimal or a binary number. With no addition, the assembler will treat the address indication as a decimal number. To indicate that the memory location is given as a hexadecimal or a binary number, one has to add a ‘h’ or a ‘b’ at the end of the address.

To define a byte, the DB (define byte) instruction can be used. The instruction:

```
newbyte:   DB 1 ;define a byte
```

defines a byte in some free location in memory. The name `newbyte` is a reference to the memory location of the newly defined byte which contains value 1.

There are four addressing modes that will be discussed in this subsection.

**Immediate addressing**

To store a value direct in some memory location, immediate addressing can be used. The instruction:

```
MOV ACC,#20h ;store 20h in A
```

can be used to store the hexadecimal value 20 in the accumulator. Instead of using ‘ACC’, the address of the accumulator in memory (E0h) can be used.

To move, for example, the value 2 to the newly defined byte `newbyte`, the following instruction can be used:

```
MOV newbyte, #2
```

**Direct addressing**

To store a value in some memory location that is obtained by directly retrieving it from another memory location the direct addressing method can be used. The instruction:

```
MOV ACC,30h ;store the value of 30h in A
```

can be used to load the value, stored at address 30h into the accumulator. Direct addressing is generally fast since, although the value to be loaded is not included in the instruction, it is quickly accessible because it is stored in internal RAM. It is more flexible than immediate addressing since the value at address 30h can be variable.

**Indirect addressing**

The instruction:

```
MOV ACC,@R0
```
B. Assembly language programming

will analyze the R0 register. The accumulator will then be loaded with the value found at
the address indicated by R0. If R0 holds the value 40h and internal RAM address 40h holds
value 60h, the above instruction will load the accumulator with the value found at address
40h. So the accumulator ends up holding the value 60h

External addressing
External memory can be addressed using another instruction in stead of the MOV instruction.
To address external RAM, the MOVX instruction has to be used. To address external code
memory, the MOVC instruction has to be used. Note that the external code memory is read
only.

To read from and write to the external RAM, the following instructions can be used:

\[
\begin{align*}
\text{MOVX} & \quad \text{ACC, @RO} \\
\text{MOVX} & \quad @\text{RO}, \text{ACC}
\end{align*}
\]

The first instruction will load the value stored at the address specified in R0 into the accu-
mulator and the second instruction will store the value of the accumulator into the address
specified by R0. To read from the external code memory the following instruction can be
used:

\[
\text{MOVC} \quad \text{ACC, @RO}
\]

Bit variables
The 8051 gives the user the ability to access a number of bit variables in the bit memory.
As mentioned before, the bit memory of the 8051 consists of 128 bits that can be accessed
separately. A bit can be either 1 or 0. To set a bit (i.e. set the value of the bit to 1), the
SETB instruction can be used. To clear a bit, the CLR instruction can be used. Some of the
special function registers are bit addressable also.

B.2 Program flow

To keep track of which instruction has to be performed, a SFR called the program counter
(PC) is used. The PC is a two byte (16 bit) address that tells the microcontroller where the
next instruction to execute is found in memory. On startup, the PC is set to 0000h. The
microcontroller begins to execute instructions sequentially in memory by incrementing the
PC after each instruction. There are several instructions that can be used to modify the PC
and that cause the program flow to deviate from its, otherwise sequential scheme. Those
instructions can be classified as:

- Conditional branching;
- Direct jumps and calls;
- Returns from subroutines and interrupts.
To change the starting address of the program, the following instruction can be used at the beginning of a program:

```
ORG 0010h ;start the program at 10h
```

This will cause the program counter to be set to 0010h. Then the program to start above 10h, so the first 16 bytes will not be used during program flow. These bytes can than be used to store variables for example.

**Conditional branching**

Conditional branching instructions cause the program execution to follow a non sequential path if a certain condition is true. The CJNE instruction for example means "compare two values and jump if not equal". The following instruction:

```
JCNE ACC, #1, next
```

next:

will check if the accumulator holds the decimal value of one. If not, the program will jump to the address named next where operation2 will be executed. Operation1 will be omitted. If the accumulator holds decimal value one, the program will continue sequentially. There are several conditional branching instructions available. A complete listing of the instruction set can be found in Appendix C. All the decisions in a program have to be made using these conditional branching instructions. Conditional branching can be thought of as the if and then statements that can be used in higher level programming languages.

**Direct jumps and calls**

To make a direct branch to a given memory location without basing it on a logical decision, one can use "direct jump" and "call" instructions. A direct jump causes the program to jump directly to the specified address. The LJMP instruction defines a long jump to a memory location. The following instruction would cause the program to jump to the address named 'address'.

```
L JMP address ;jump to address named 'address'
```

There are two other instructions that have the same effect. The SJMP (short jump) and AJMP (absolute jump) can be used to jump to another memory location, there is a restriction however to the locations the program can jump to. On the other hand SJMP and AJMP require only two bytes of code, while the LJMP instruction requires 3 bytes. A good assembler often takes care of replacing long jumps by a small jump or an absolute jump if possible.

Another powerful operation is the call to a sub-routine. The LCALL (long call) instruction causes the program to jump to a specified address. On that address a sub-routine starts that ends with the RET (return) instruction. The RET instruction then causes the program to jump back to the address following the LCALL instruction that called the sub-routine. The following code sample illustrates the use of LCALL and RET.
B. Assembly language programming

```
LCALL sub_routine
operation1
sub_routine: operation2
RET
```

This sample code will cause the program to jump to the address named `sub_routine`. After executing `operation2`, the program will return to the location after the LCALL instruction and `operation1` will be executed.

The program flow can also be altered by interrupts.
B. Assembly language programming
Appendix C

The 80C32 Instruction set

Notes on instruction set and addressing modes:

Rn  Register R7-R0 of the currently selected Register Bank.
direct  8-bit internal data locations address.
@Ri  8-bit internal data RAM location (0-255),
     addressed indirectly through register R1 or R0.
#data  8-bit constant included in the instruction.
#data 16  16-bit constant included in the instruction
addr 16  16-bit destination address. Used by LCALL and LJMP.
addr 11  11-bit destination address. Used by ACALL and AJMP.
rel  Signed (two's complement) 8-bit offset byte.
     Used by SJMP and all conditional jumps.
     Range is 128 to +127 bytes relative to first byte of the following instruction.
bit  Direct Addressed bit in Internal Data RAM or Special Function Register.

Table C.1: The 80C32 instruction table

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>A,Rn</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>A,direct</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>A,&lt;@Ri</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>A,#data</td>
<td>1</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,Rn</td>
<td>1</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,direct</td>
<td>1</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,#data</td>
<td>1</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,@Ri</td>
<td>1</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,#data</td>
<td>1</td>
</tr>
</tbody>
</table>

41
## C. The 80C32 Instruction set

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBB A,Rn</td>
<td>Subtract Register from ACC with borrow</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,direct</td>
<td>Subtract direct byte from ACC with borrow</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,@Ri</td>
<td>Subtract indirect RAM from ACC with borrow</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,#data</td>
<td>Subtract immediate data from ACC with borrow</td>
<td>1</td>
</tr>
<tr>
<td>INC A</td>
<td>Increment Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>INC Rn</td>
<td>Increment register</td>
<td>1</td>
</tr>
<tr>
<td>INC direct</td>
<td>Increment direct byte</td>
<td>1</td>
</tr>
<tr>
<td>INC @Ri</td>
<td>Increment indirect RAM</td>
<td>1</td>
</tr>
<tr>
<td>DEC A</td>
<td>Decrement Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>DEC Rn</td>
<td>Decrement Register</td>
<td>1</td>
</tr>
<tr>
<td>DEC direct</td>
<td>Decrement direct byte</td>
<td>1</td>
</tr>
<tr>
<td>DEC @Ri</td>
<td>Decrement indirect RAM</td>
<td>1</td>
</tr>
<tr>
<td>INC DPTR</td>
<td>Increment Data Pointer</td>
<td>2</td>
</tr>
<tr>
<td>MUL AB</td>
<td>Multiply A and B</td>
<td>4</td>
</tr>
<tr>
<td>DIV AB</td>
<td>Divide A by B</td>
<td>4</td>
</tr>
<tr>
<td>DA A</td>
<td>Decimal Adjust Accumulator</td>
<td>1</td>
</tr>
</tbody>
</table>

### Logical Operations

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANL A,Rn</td>
<td>AND Register to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ANL A,direct</td>
<td>AND direct byte to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ANL A,@Ri</td>
<td>AND indirect RAM to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ANL A,#data</td>
<td>AND immediate data to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ANL direct,A</td>
<td>AND Accumulator to direct byte</td>
<td>1</td>
</tr>
<tr>
<td>ANL direct,#data</td>
<td>AND immediate data to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>ORL A,Rn</td>
<td>OR register to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ORL A,direct</td>
<td>OR direct byte to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ORL A,@Ri</td>
<td>OR indirect RAM to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ORL A,#data</td>
<td>OR immediate data to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ORL direct,A</td>
<td>OR Accumulator to direct byte</td>
<td>1</td>
</tr>
<tr>
<td>ORL direct,#data</td>
<td>OR immediate data to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>XRL A,Rn</td>
<td>Exclusive-OR register to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XRL A,direct</td>
<td>Exclusive-OR direct byte to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XRL A,@Ri</td>
<td>Exclusive-OR indirect RAM to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XRL A,#data</td>
<td>Exclusive-OR immediate data to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XRL direct,A</td>
<td>Exclusive-OR Accumulator to direct byte</td>
<td>1</td>
</tr>
<tr>
<td>XRL direct,#data</td>
<td>Exclusive-OR immediate data to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>CLR A</td>
<td>Clear Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>CPL A</td>
<td>Complement Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>RL A</td>
<td>Rotate Accumulator left</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Accumulator left through the carry</td>
<td>1</td>
</tr>
<tr>
<td>RR A</td>
<td>Rotate Accumulator right</td>
<td>1</td>
</tr>
<tr>
<td>RRC A</td>
<td>Rotate Accumulator right through the carry</td>
<td>1</td>
</tr>
<tr>
<td>SWAP A</td>
<td>Swap nibbles within the Accumulator</td>
<td>1</td>
</tr>
</tbody>
</table>

### Data Transfer

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV A,Rn</td>
<td>Move register to Accumulator</td>
<td>1</td>
</tr>
</tbody>
</table>
### C. The 80C32 Instruction set

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV A,direct</td>
<td>Move direct byte to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>MOV A,@Ri</td>
<td>Move indirect RAM to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>MOV A,#data</td>
<td>Move immediate data to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>MOV Rn,A</td>
<td>Move Accumulator to register</td>
<td>1</td>
</tr>
<tr>
<td>MOV Rn,direct</td>
<td>Move direct byte to register</td>
<td>2</td>
</tr>
<tr>
<td>MOV RN,#data</td>
<td>Move immediate data to register</td>
<td>1</td>
</tr>
<tr>
<td>MOV direct,A</td>
<td>Move Accumulator to direct byte</td>
<td>1</td>
</tr>
<tr>
<td>MOV direct,Rn</td>
<td>Move register to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>MOV direct,direct</td>
<td>Move direct byte to direct</td>
<td>2</td>
</tr>
<tr>
<td>MOV direct,@Ri</td>
<td>Move indirect RAM to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>MOV direct,#data</td>
<td>Move immediate data to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>MOV @Ri,A</td>
<td>Move Accumulator to indirect RAM</td>
<td>1</td>
</tr>
<tr>
<td>MOV @Ri,direct</td>
<td>Move direct byte to indirect RAM</td>
<td>2</td>
</tr>
<tr>
<td>MOV @Ri,#data</td>
<td>Move immediate data to indirect RAM</td>
<td>1</td>
</tr>
<tr>
<td>MOV DPTR,#data16</td>
<td>Load Data Pointer with a 16-bit constant</td>
<td>2</td>
</tr>
<tr>
<td>MOV C,A+DPTR</td>
<td>Move Code byte relative to DPTR to ACC</td>
<td>2</td>
</tr>
<tr>
<td>MOV C,A+PC</td>
<td>Move Code byte relative to PC to ACC</td>
<td>2</td>
</tr>
<tr>
<td>MOVX A,@Ri</td>
<td>Move external RAM (8-bit addr) to ACC</td>
<td>2</td>
</tr>
<tr>
<td>MOVX A,DPTR</td>
<td>Move external RAM (16-bit addr) to ACC</td>
<td>2</td>
</tr>
<tr>
<td>MOVX A,@Ri,A</td>
<td>Move ACC to external RAM (8-bit addr)</td>
<td>2</td>
</tr>
<tr>
<td>MOVX @DPTR,A</td>
<td>Move ACC to external RAM (16-bit addr)</td>
<td>2</td>
</tr>
<tr>
<td>PUSH direct</td>
<td>Push direct byte onto stack</td>
<td>2</td>
</tr>
<tr>
<td>POP direct</td>
<td>Pop direct byte from stack</td>
<td>2</td>
</tr>
<tr>
<td>XCH A,Rn</td>
<td>Exchange register with Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XCH A,direct</td>
<td>Exchange direct byte with Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XCH A,@Ri</td>
<td>Exchange indirect RAM with Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XCHD A,@Ri</td>
<td>Exchange low-order digit indirect RAM with ACC</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Boolean Variable Manipulation

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR C</td>
<td>Clear carry</td>
<td>1</td>
</tr>
<tr>
<td>CLR bit</td>
<td>Clear direct bit</td>
<td>1</td>
</tr>
<tr>
<td>SETB C</td>
<td>Set carry</td>
<td>1</td>
</tr>
<tr>
<td>SETB bit</td>
<td>Set direct bit</td>
<td>1</td>
</tr>
<tr>
<td>CPL C</td>
<td>Complement carry</td>
<td>1</td>
</tr>
<tr>
<td>CPL bit</td>
<td>Complement direct bit</td>
<td>1</td>
</tr>
<tr>
<td>ANL C,bit</td>
<td>AND direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>ANL C,/bit</td>
<td>AND complement of direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>ORL C,bit</td>
<td>OR direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>ORL C,/bit</td>
<td>OR complement of direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>MOV C,bit</td>
<td>Move direct bit to carry</td>
<td>1</td>
</tr>
<tr>
<td>MOV bit,C</td>
<td>Move carry to direct bit</td>
<td>2</td>
</tr>
<tr>
<td>JC rel</td>
<td>Jump if carry is set</td>
<td>2</td>
</tr>
<tr>
<td>JNC rel</td>
<td>Jump if carry not set</td>
<td>2</td>
</tr>
<tr>
<td>JB rel</td>
<td>Jump if direct bit is set</td>
<td>2</td>
</tr>
<tr>
<td>JNB rel</td>
<td>Jump if direct bit is not set</td>
<td>2</td>
</tr>
</tbody>
</table>
### C. The 80C32 Instruction set

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACALL</td>
<td>Absolute subroutine call</td>
<td>2</td>
</tr>
<tr>
<td>LCALL</td>
<td>Long subroutine call</td>
<td>2</td>
</tr>
<tr>
<td>RET</td>
<td>Return from subroutine</td>
<td>2</td>
</tr>
<tr>
<td>RETI</td>
<td>Return from interrupt</td>
<td>2</td>
</tr>
<tr>
<td>AJMP</td>
<td>Absolute jump</td>
<td>2</td>
</tr>
<tr>
<td>LJMP</td>
<td>Long jump</td>
<td>2</td>
</tr>
<tr>
<td>SJMP</td>
<td>Short jump (relative addr)</td>
<td>2</td>
</tr>
<tr>
<td>JMP @A+DPTR</td>
<td>Jump indirect relative to the DPTR</td>
<td>2</td>
</tr>
<tr>
<td>JZ rel</td>
<td>Jump if Accumulator is zero</td>
<td>2</td>
</tr>
<tr>
<td>JNZ rel</td>
<td>Jump if Accumulator is not zero</td>
<td>2</td>
</tr>
<tr>
<td>CJNE A,direct,rel</td>
<td>Compare direct byte to ACC and jump if not equal</td>
<td>2</td>
</tr>
<tr>
<td>CJNE A,#data,rel</td>
<td>Compare immediate to ACC and jump if not equal</td>
<td>2</td>
</tr>
<tr>
<td>CJNE RN,#data,rel</td>
<td>Compare immediate to register and jump if not equal</td>
<td>2</td>
</tr>
<tr>
<td>CJNE @Ri,#data,rel</td>
<td>Compare immediate to indirect and jump if not equal</td>
<td>2</td>
</tr>
<tr>
<td>DJNZ Rn,rel</td>
<td>Decrement register and jump if not zero</td>
<td>2</td>
</tr>
<tr>
<td>DJNZ direct,rel</td>
<td>Decrement direct byte and jump if not zero</td>
<td>2</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>1</td>
</tr>
</tbody>
</table>

**Program Branching**

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>description</th>
<th># of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>JBC bit,rel</td>
<td>Jump if direct bit is set and clear bit</td>
<td>2</td>
</tr>
</tbody>
</table>

**Program Flow**

- **Jumping:**
  - ACALL, LCALL: Subroutine calls
  - RET, RETI: Return from subroutine
  - AJMP, LJMP: Absolute jump
  - SJMP, JMP @A+DPTR: Jump to a location
  - CJNE: Compare and jump if not equal
  - DJNZ: Decrement and jump if not zero

- **Branching:**
  - JBC: Jump if bit is set and clear

- **No Operation:**
  - NOP: No operation
Appendix D

Windows DLL files

D.1 command.cpp

Command.cpp is the source listing of the DLL file that includes the routines that are used in communicating with Trilobot in command mode. The names of these routines start with 'Cmd'.

#define STRICT
#define MAX 1024;
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

static double OUTPUT;
static int TERM, COMWRITE, COMREAD;
char *port = "com1";
char msg[24];
unsigned long n_in, n_out;
BOOL fSuccess;
HANDLE hPort;
DCB dcb;
FILE *fidda;

BOOL APIENTRY DllMain( HANDLE hModule,
                        DWORD ul_reason_for_call,
                        LPVOID lpReserved

                        )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

static DWORD TRILOThreadID;
DWORD TRILOThread(LPVOID);
extern "C" _declspec(dllexport) int CmdTRILOStart(void)
{
    CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)TRILOThread,
        (LPVOID) NULL,0,&TRILOThreadID);
    return 0;
}

extern "C" _declspec(dllexport) int CmdTRILOOpen(void)
{
    hPort = CreateFile(port, GENERIC_READ|GENERIC_WRITE,
        0, NULL,
        TRUNCATE_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL);

    if (hPort == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d.\n", GetLastError());
        return 1;
    }

    fSuccess = GetCommState(hPort, &dcb);
    if (!fSuccess)
    {
        printf("GetCommState failed with error %d.\n", GetLastError());
        return 2;
    }

    dcb.BaudRate = CBR_9600;
    dcb.ByteSize = 8;
    dcb.Parity = NOPARITY;
    dcb.StopBits = ONESTOPBIT;

    if (fSuccess)
    {
        printf("Serial port %s (re)configured: BaudRate %d, ByteSize %d, Parity %d.\n",
            port,dcb.BaudRate,dcb.ByteSize,dcb.Parity);

        COMMTIMEOUTS timeouts;
        timeouts.ReadIntervalTimeout = 0;
        timeouts.ReadTotalTimeoutMultiplier = 0;
        timeouts.ReadTotalTimeoutConstant = 0;
        timeouts.WriteTotalTimeoutMultiplier = 0;
        timeouts.WriteTotalTimeoutConstant = 0;

        fSuccess = SetCommTimeouts(hPort, &timeouts);
        if (!fSuccess)
        {
            printf("SetCommTimeouts failed with error %d.\n", GetLastError());
            return 3;
        }

        return 1;
    }
}

DWORD TRILOThread(LPVOID param)
{
D. Windows DLL files

TERM = 1;

fSuccess = PurgeComm(hPort, PURGE_RXCLEAR);
if (! fSuccess)
{
    printf ("Purge inputbuffer failed with error %d.\n", GetLastError());
    return (3);
}

fSuccess = PurgeComm(hPort, PURGE_TXCLEAR);
if (! fSuccess)
{
    printf ("Purge outputbuffer failed with error %d.\n", GetLastError());
    return (3);
}

TERM = 0;

while (TERM==0)
{
    //declare vars
    n_in=0;
    n_out=0;
    int msg1, msg2;
    strcpy (msg, "aa");

    CMMWRITE = WriteFile(hPort, "!GF1", 6, & n_in, NULL);
    if (! CMMWRITE)
    {
        printf ("Writing to port %s failed: error -> %d \n", port, GetLastError());
    }

    CMMREAD = ReadFile(hPort, msg, 2, & n_out, NULL);

    msg1 = msg[0];
    if(msg1 < 65){msg1 = msg1 - 48;}
    if(msg1 >= 65){msg1 = msg1 - 54;}

    msg2 = msg[1];
    if(msg2 < 65){msg2 = msg2 - 48;}
    if(msg2 >= 65){msg2 = msg2 - 54;}

    OUTPUT = 16*msg1 + msg2;
}

fSuccess = CloseHandle(hPort);
if (! fSuccess)
{
    printf ("CloseHandle failed with error %d.\n", GetLastError());
    return (4);
}

if (fSuccess)
{
    printf ("The serial communication has been stopped");
}
return 0;

extern "C" _declspec(dllexport) int CmdTRILOutput(double* Get_Output)
{
    Get_Output[0]=OUTPUT;
    return 0;
}

extern "C" _declspec(dllexport) int CmdTRILTerm(void)
D. Windows DLL files

{  
   TERM = 1;
   return 0;
}

extern "C" _declspec(dllexport) int CmdTRILODstatus(void)
{  
   if (!COMWRITE)
      {return 0;}  
   else
      {return 1;}  
}

D.2 trilo.cpp

Trilo.cpp is the source listing of the DLL file that includes the routines that are used in communicating with Trilobot when Trilobot is running the user program listed in Appendix F.

#define STRICT
#define MAX 1024;

#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <tio1ib.h>
#include <time.h>

static double DMDIST, PIR;
static int TERM, COMWRITE, COMREAD;
char *port = "com1";
char msg[6];
unsigned long n_in, n_out;
HANDLE hPort;
DCB dcb;

BOOL APIENTRY DllMain(HANDLE hModule,
 DWORD ul_reason_for_call,
LPVOID lpReserved)
{  
   switch (ul_reason_for_call)
   {  
      case DLL_PROCESS_ATTACH:
      case DLL_THREAD_ATTACH:
      case DLL_THREAD_DETACH:
      case DLL_PROCESS_DETACH:
         break;
   }  
   return TRUE;
}

static DWORD TRILOThreadID;
DWORD TRILOThread(LPVOID);

extern "C" _declspec(dllexport) void TRILOStart(void)
{  
   CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)TRILOThread,
   (LPVOID) NULL,0,&TRILOThreadID);
}

extern "C" _declspec(dllexport) void TRILOOpen(void)
{  
   
48
D. Windows DLL files

hPort = CreateFile(port, GENERIC_READ|GENERIC_WRITE, 0, NULL, TRUNCATE_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

dcb.BaudRate = CBR_9600;
dcb.ByteSize = 8;
dcb.Parity = NOPARITY;
dcb.StopBits = ONESTOPBIT;

COMMTIMEOUTS timeouts;
timeouts.ReadIntervalTimeout = 0;
timeouts.ReadTotalTimeoutMultiplier = 0;
timeouts.ReadTotalTimeoutConstant = 0;
timeouts.WriteTotalTimeoutMultiplier = 0;
timeouts.WriteTotalTimeoutConstant = 0;

if (hPort == INVALID_HANDLE_VALUE)
{
  printf ("CreateFile failed with error %d.\n", GetLastError());
}

if (!GetCommState(hPort, &dcb))
{
  printf ("GetCommState failed with error %d.\n", GetLastError());
}

if (!SetCommState(hPort, &dcb))
{
  printf ("SetCommState failed with error %d.\n", GetLastError());
}
else
{
  printf("Serial port %s (re)configured: BaudRate %d, ByteSize %d, Parity %d,\n", port, dcb.BaudRate, dcb.ByteSize, dcb.Parity);
}

if (!SetCommTimeouts(hPort, &timeouts))
{
  printf ("SetCommTimeouts failed with error %d.\n", GetLastError());
}

DWORD TRILOTHread(LPVOID param)
{
  TERM = 0;

  while (TERM==0)
  {
    n_in=0;
    n_out=0;
    int msg1, msg2, msg3, msg4, msg5, msg6;
    strcpy(msg, "aaaaaa");
    COMREAD = ReadFile(hPort, msg, 6, &n_out, NULL);
    msg1 = msg[0];
    if(msg1 < 65){msg1 = msg1 - 48;}
    if(msg1 >= 65){msg1 = msg1 - 54;}
    msg2 = msg[1];
if (msg2 < 65) \{msg2 = msg2 - 48;\}
if (msg2 >= 65) \{msg2 = msg2 - 54;\}
msg3 = msg[2];
if (msg3 < 65) \{msg3 = msg3 - 48;\}
if (msg3 >= 65) \{msg3 = msg3 - 54;\}
msg4 = msg[3];
if (msg4 < 65) \{msg4 = msg4 - 48;\}
if (msg4 >= 65) \{msg4 = msg4 - 54;\}
msg5 = msg[4];
if (msg5 < 65) \{msg5 = msg5 - 48;\}
if (msg5 >= 65) \{msg5 = msg5 - 54;\}
msg6 = msg[5];
if (msg6 < 65) \{msg6 = msg6 - 48;\}
if (msg6 >= 65) \{msg6 = msg6 - 54;\}

DMDIST = 4096*msg1 + 256*msg2 + 16*msg3 + msg4;
PIR = 16*msg5 + msg6;
}

if (!CloseHandle(hPort))
{
    printf("CloseHandle failed with error %d.\n", GetLastError());
}
else
{
    printf("The serial communication has been stopped");
}
return 0;

extern "C" _declspec(dllexport) void TRILOPir(double* Get_Pir)
{
    Get_Pir[0]=PIR;
}
extern "C" _declspec(dllexport) void TRILODMDist(double* Get_DMDist)
{
    Get_DMDist[0]=DMDIST;
}

extern "C" _declspec(dllexport) void TRILOTerm(void)
{
    TERM = 1;
}

extern "C" _declspec(dllexport) void TRILOSend(void)
{
    COMWRITE = WriteFile(hPort,"!",2,&n_in, NULL);
    if (!COMWRITE)
        printf("Writing to port %s failed: error -> %d \n",port,GetLastError());
}
Appendix E

Simulink S-functions

E.1  s_trilo.c

#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME s_trilo

#include "simstruc.h"
#include <string.h>
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

extern int CmdTRILOSTart(void);
extern int CmdTRILOOutput(double*);
extern int CmdTRILOTerm(void);
extern int CmdTRILOSstatus(void);

static voidmdlInitializeSizes(SimStruct *S)
{
    #ifndef MATLAB_MEX_FILE
        printf("Starting TRILO system...\n");
        CmdTRILOSopen();
        //TRILOClear();
        CmdTRILOSTart();
        while (!CmdTRILOSstatus())
        {
            printf(".");
            Sleep(100);
        }
        printf("nTRIL0 system initialised.n");
    #endif
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
    {
        return;
    }
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);
    if (!ssSetNumInputPorts(S, 0)) return;
    if (!ssSetNumOutputPorts(S, 1)) return;
    printf("TRILO system initialised\n");
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
    {
        return;
    }
}
E. Simulink S-functions

```c
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
    double hoek;
    real_T *hoeklong = ssGetOutputPortRealSignal(S, 0);
    #ifndef MATLAB_MEX_FILE
    CmdTRILOOutput(&hoek);
    hoeklong[0] = hoek;
    #endif
}

static void mdlTerminate(SimStruct *S)
{
    #ifdef MATLAB_MEX_FILE
    CmdTRILOTerm();
    #endif
}

#include "simstruc.h"
#define NSTATES 0
#define NINPUTS 1
#define NOUTPUTS 0
#define NPARAMS 0
void TRILOSend(void);

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumContStates(S, NSTATES);
    ssSetNumDiscStates(S, 0);
}
```

---

**E.2 s_trilo_in.c**

```c
#define S_FUNCTION_NAME s_trilo_in
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

#define NSTATES 0
#define NINPUTS 1
#define NOUTPUTS 0
#define NPARAMS 0

#define U (*uPtrs)

#include <math.h>

void TRILOSend(void);

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumContStates(S, NSTATES);
    ssSetNumDiscStates(S, 0);
}
```

52
if (!ssSetNumInputPorts(S,l)) return;
ssSetInputPortWidth(S,0,NINPUTS);
ssSetInputPortDirectFeedThrough(S,0,0);

if (!ssSetNumOutputPorts(S,0)) return;
ssSetNumSampleTimes(S,1);
ssSetNumNumWork(S,0);
ssSetNumNumWork(S,0);
ssSetNumNumModes(S,0);
ssSetNumNonsampledZCs(S,0);
}

static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S,0,CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S,0,0.0);
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
    InputRealPtrsType uPtrs=ssGetInputPortRealSignalPtrs(S,0);

#if defined MATLAB_MEX_FILE
    if(*U != 0)
    {
        TRILOSend();
    }
#endif
}

static void mdlTerminate(SimStruct *S)
{
}

#if defined MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif

E.3  s_trilo_out.c

#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME  s_trilo_out

#include "simstruc.h"
#include <string.h>
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

extern void TRILOStart(void);
extern void TRILOOpen(void);
extern void TRILOPir(double*);
extern void TRILOMDist(double*);
extern void TRILOTerm(void);

static void mdlInitializeSizes(SimStruct *S)
{
#if defined MATLAB_MEX_FILE
    printf("\nStarting TRILO system...\n");


TRILOCpen();
printf("nTRILO system initialised.");
TRILOSTart();
#endif

ssSetNumSFcnParams(S, 0);
if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
    return;
}

ssSetNumContStates(S, 0);
ssSetNumDiscStates(S, 0);
if (!ssSetNumInputPorts(S, 0)) return;
if (!ssSetNumOutputPorts(S, 2)) return;
ssSetInputPortWidth(S, 0, 1);
ssSetInputPortWidth(S, 1, 1);
ssSetNumOutputPorts(S, 1);
ssSetNumWork(S, 0);
ssSetNumIWork(S, 0);
ssSetNumFWork(S, 0);
ssSetNumNodes(S, 0);
ssSetNumNonSampledZCs(S, 0);
ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

static void mdlInitializeSampleTimes(SimStruct *S) {
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

static void mdlOutputs(SimStruct *S, int_T tid) {
    double pir;
    real_T *pirlong = ssGetOutputPortRealSignal(S,0);
    double dmdist;
    real_T *dmdistlong = ssGetOutputPortRealSignal(S,1);
#ifndef MATLAB_MEX_FILE
    TRILOPir(&pir);
    pirlong[O] = pir;
    TRILOMDist(&dmdist);
    dmdistlong[O] = dmdist;
#endif
}

static void mdlTerminate(SimStruct *S) {
#ifndef MATLAB_MEX_FILE
    TRILOTerm();
#endif
}

#endif ndef MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif

54
Appendix F
Userprogram.a51

$include(trilodef.a51) ;include predefined routines
#include(R5GB51.inc) ;include SFR addresses

org 0100h ;start program above interrupt vectors

init:  lcall trmsclr ;clear LCD.
lcall check_batt ;check battery
mov A, #2
lcall utlwas

pause:  lcall trmsclr ;clear LCD
lcall sonint ;initialize sonar
lcall sioint ;initialize serial port
lcall LCD_start ;display start message
lcall wait_key ;wait key press
lcall utlhloff ;headlight off
lcall LCD_run ;display run message
ljmp start ;start loop

;main loop

start:  lcall check_key ;check keypresses
lcall check_in ;check serial communication
lcall utlgod ;get drive motor distance
lcall siophw ;write dmdist to serial port
lcall sioget ;if so, get sonar
lcall siophb ;write sonar to serial port
mov A, #50
lcall utlwam
lcall utlgd
jnb ACC.0, wait-Ykey ;if so, pause on whisker detect
loop:  ljmp start ;loop

;wait for keypress

wait_key:  lcall trmsgs ;get keypad status
cjne A, #'1', wait_Ykey ;check if key 1 pressed
lcall help ;if so, move to display utility

wait_Ykey:  cjne A, #'Y', wait_key ;no Y keypress -> loop
ret ;return on key Y press

;startup message

LCD_start:  lcall trmsclr ;clear LCD
lcall trmsps ;Message on LCD.
db 0 ;Top line.
db 'TU/e Trilobot'
db 0
lcall trmsps ;Message on LCD.
db 40h ;Bottom line.
db 'Y=Start 1=Help'
db 0
ret

;message while running
LCD_run: lcall tmclr
lcall trmpsi
db 0
db 'Running program'
db 0
lcall trmpsi
db 0
db 'N=Quit  Y=Pause'
db 0
ret

;act on key presses
check_key: lcall trmgs
jnc ex_keys
cjne A,'#N',check_Y
lcall tmclr
lcall trmpsi
lcall siocgb
lcall siocpb
lcall sysres
check-Y: cjne A,'#Y',ex-keys
ljmp pause
lcail trmclr
ex-keys: ret

;serial check procedure
check-in: lcall siogs
jnc ex-serial
mov 0200h, A
mov A, #2
lcall utlgol
ex-serial: ret

;check whisker status
whisker: lcall utlgw
cjne A,'#00h,whisker_act
jmp ex-whisker
whisker_act: lcall utlhlon
ljmp pause
lcall trmclr
ex-whisker: ret

;check the battery on startup
check-batt: lcall utlgbs
cjne A, '#1, batt2
lcall tmclr
lcall trmpsi
db 0
db 'Battery OK'
dx 0
batt2: lcall utlgbs
cjne A, '#2, batt3
lcall tmclr
lcall trmpsi
db 0
db 'Battery low'
dx 0
batt3: lcall utlgbs
cjne A, '#3, batt4
lcall tmclr
lcall trmpsi
db 0
db 'Battery very low'

F. Userprogram.a51

```
db 0
batt4: db 0
    lcall utlgbe
    cjne A, #4, batt_ex
    lcall trmclr
    lcall trmpsi
    db 0
    db 'Battery too low'
    db 0
batt_ex: ret

;help display utility
help: lcall trmclr
    mov dptr, #help_msg
    mov A, #2
    lcall trmsg
    lcall check_hlp
check_hlp: lcall trmsg
    cjne A,#'N',hlp_N
    lcall trmclr
    lcall init
    lcall help
hlp_N: cjne A,#'l',check_hlp
    ret
hlp-1: cjne A,#'l',check_hlp
    lcall help
    ret

;message string to display with trmsg
help_msg: db 'TU/e Trilo Help',
    '**************************',
    db 'This program can',
    db 'be used iw the',
    db 'trilo interface',
    db 'in Simulink',
    db 'If dipswitch1 is',
    db 'activated, trilo',
    db 'will pause and',
    db 'turn on the head',
    db 'light if a',
    db 'whisker is on',
    db 'N=Exit  1=Back',
    '**************************',
    db 0
end
```

;clear LCD
;Message on LCD.
;Top line.
;clear LCD
;load help message in dptr
;define display rate
;display message utility
;get keypad status
;clear LCD
;key N press -> exit
;key 1 press ->replay help
;end of userprogram.a5i