Implementation of a Sonar Array for Robot Navigation

ALEXANDER TOLLET

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2012
Implementation of a Sonar Array for Robot Navigation

ALEXANDER TOLLET

2D1021, Master’s Thesis in Computer Science (30 ECTS credits)
Degree Progr. in Electrical Engineering 270 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Stefan Carlsson
Examiner was Stefan Carlsson

TRITA-CSC-E 2012:090
ISRN-KTH/CSC/E--12/090--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc
Implementation of a sonar array for robot navigation

Abstract
In this thesis, an implementation of a microprocessor based measurement system is presented. This system controls an array of sonar sensors for robot navigation. The electronics between the Nomad robot platform and the sonar array was developed, as well as the communication interface. Programs were written for both microprocessor and PC to control and configure the sonar array.

The implemented system makes it possible to coordinate multiple sonar sensors for simultaneous transmission. It also allows echo reception by multiple sensors simultaneously, rather than just the firing sensor. This provides the significant advantage of being able to compute accurate distance and bearing information from only one measurement session.

Implementering av en sonarmatris för robotnavigation

Sammanfattning
I detta examensarbete presenteras en implementering av ett mikroprosessor baserat mätsystem som styr en matris av sonarsensorer för robotnavigering. Elektroniken mellan Nomad robotplattformen och sonarmatrisen utvecklades, liksom kommunikationsgränssnittet. Programkod som styr och konfigurerar sonarmatrisen skrevs för både mikroprocessor och PC.

Det implementerade systemet gör det möjligt att koordinera multipla sensorer för simultan signaltransmission. Det möjliggör även ekoreception från multipla sensorer samtidigt, istället för bara av den sensor som sänder ut signalen. Detta ger den avgörande fördelen att korrekt avstånds- och riktningsinformation kan erhållas vid ett enda mättilfälle.
Acknowledgments

This thesis was written to fulfil the requirement for the award of M.Sc. in engineering at the school of Computer Science and Communication (CSC) at Kungliga Tekniska Högskolan (KTH). The research was performed at CSC under the supervision of Henrik Christiansen.

First of all I would like to dedicate this work to my son Kevin Tollet who has proven to me that no matter how wondrous the world of robotics seems, there is still nothing more amazing than a brand new human being. It is my hope that Kevin’s generation is the one that will truly experience a world where robots are an integral and helpful part of human society.

I would like to acknowledge a number of people for providing valuable assistance towards the completion of this work: Henrik Christensen, my supervisor, for assisting with the specification of the functionality requirements and for providing the necessary funding to build the sonar array research platform; Johan Gullman-Strand for helpful technical discussions and mathematical input; Mikael Naeslund for helping me get started in writing the thesis; and Olle Wijk for using the completed sonar array in his research and helping me prove that I have created a useful and functional research tool for CAS.

I would also like to thank the people who have helped me structure and proofread the thesis: Yufeng Zhao, Jenny Tollet and Mark Lawson.

Finally, I want to thank my wife Yufeng for the wonderful support she has provided me throughout this project.
1 Introduction

1.1 Robots and their applications

1.2 Purpose

1.3 Outline

2 Overview of sonar sensors

2.1 Sensing the environment using sonar

3 Original system

3.1 System specification

3.2 Polaroid 6500 series sonar ranging module
   3.2.1 Control interface

3.3 Estimating position by triangulation

3.4 Weaknesses

4 Goals, requirements and limitations

4.1 General goal

4.2 Assumptions regarding the operating domain
   4.2.1 Indoor environment
   4.2.2 Choice of sensor

4.3 Requirements

5 Solution

5.1 Basic approach

5.2 Position estimation by multiple receptions

5.3 System overview and architecture

6 Hardware platform design

6.1 The Polaroid 6500 sonar sensor
   6.1.1 Operating requirements
   6.1.2 Performance and accuracy

6.2 Modifications of the sensor electronics

6.3 Sensor controller

PIC specification
   6.3.2 Mux specification
   6.3.3 PC-PIC interface

7 Software design
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 PC software</td>
<td>..................................................................................................</td>
<td>23</td>
</tr>
<tr>
<td>7.2 Sonar array controller</td>
<td>..................................................................................................</td>
<td>23</td>
</tr>
<tr>
<td>8 Conclusion and future work</td>
<td>..................................................................................</td>
<td>25</td>
</tr>
<tr>
<td>9 References</td>
<td>..................................................................................................</td>
<td>27</td>
</tr>
<tr>
<td>Appendix 1: Mux Card</td>
<td>..................................................................................................</td>
<td>28</td>
</tr>
<tr>
<td>Appendix 2: PIC Card</td>
<td>..................................................................................................</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>PIC Card Oscillator</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>PIC Card PIC</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>PIC Card RS232</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>PIC Card Power</td>
<td>32</td>
</tr>
<tr>
<td>Appendix 3: Sonar_array.h</td>
<td>.................................................................................</td>
<td>33</td>
</tr>
<tr>
<td>Appendix 4: Sonar_array.cc</td>
<td>.....................................................................................</td>
<td>37</td>
</tr>
<tr>
<td>Appendix 5: Sonar_array_main.cc</td>
<td>........................................................................</td>
<td>48</td>
</tr>
<tr>
<td>Appendix 6: Sonar.c</td>
<td>..................................................................................................</td>
<td>49</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Robots and their applications

Humans are continually striving to have more comfortable lives. One avenue to obtain this goal has been to produce products and services efficiently, that require as little human effort as possible. To date, humans have been successful in manufacturing products using machines with various degrees of automation. These machines are usually confined to a factory floor with a static or controlled environment, and are built accordingly.

The next step in our mechanical evolution is to build machines that operate independently, in ordinary society, to serve humans in their daily activities. It is likely to be only a matter of time until machines will perform the work we don’t want to do. These machines will clean our homes, produce and cook our food, and perform a myriad of other useful tasks.

To be able to do these tasks, intelligent machines will need to adapt to the complexity of the human everyday environment, as well as being capable of making their own decisions. This type of machine is generally called an autonomous robot, and research into their development is currently taking place.

At the time of this research, the functionality of autonomous robots was still at a fairly basic stage. One of the key requirements at the time was to provide robots with the ability to navigate through their environment. It is only after the robot can assess its current position, the goal position, and understand the environment in between, that it can then process a route for travelling. Only then can the robot perform the task required of it.

It is a trivial task for most humans to navigate through the environment of our apartment, from the bed to the mailbox, in order to pick up the newspaper. But bear in mind, it required natural selection over billions of years to produce this functionality. We are now trying to provide machines with this gift in less than a fraction of that time. The general navigation functionality problem that must be solved by these machines can be summarised by three questions:

1. Where am I?
2. Where am I going?
3. How do I get there?

To be able to answer these questions, the minimum requirement is to gain sufficient knowledge about the operating environment. Only when the robot identifies the distance and bearing of obstacles, can it begin processing an understanding of the immediate environment. With this information, the robot can then process a path towards the goal, with obstacle avoidance strategy integrated into the route.

A large component of the above problem will require robots to sense the environment on a continual basis. Humans and animals use many senses to obtain information about the environment. Vision, touch, and hearing are the three main senses utilised. Likewise, robots use sensors, usually transducers, to learn about their operating environment.
1.2 Purpose

The purpose of this work was to develop and implement a measurement system that uses sonar array on a robot to measure both the distance and a more accurate angle to the object in its environment than the system currently employed at CAS. The goal was to construct a flexible system to be used primarily for research purposes. The intention was that the system would be used to determine the optimal sonar array to obtain both distance and angle measurements between the robot and the objects in its surroundings.

This goal was to be reached by configuring an array of sonar sensors in such a way that they can cooperate. The sensors had to be placed in a suitable geometric formation. Several transducers can be selected to transmit at same time, and the echo can be detected by several transducers. Depending on the geometrical placement of the receiving transducers, varying distances to the object can be computed. From this data, the distance between the robot and the object, and the relative angle to the object from the robot can be derived.

The significant advantage of the new configuration is that only one measurement session is required to obtain multiple measurements. These can then be used to accurately calculate an object’s position, while at the same time minimising processing time.

1.3 Outline

The basic theory for sonar measurement is given in chapter 2.

In chapter 3 the original system used at CAS is described. The Polaroid 6500 sonar control interface is studied and its weaknesses are pointed out. The triangulation method to estimate position is also introduced in this chapter.

The scope and requirements of this project is specified in chapter 4, as well as the operating assumptions.

Chapter 5 is an overview of the solution. Basic approach, system architecture and a simple algorithm are described in this chapter.

In chapters 6 and 7 the implemented hardware (HW) and software (SW) is described in detail.

In chapter 8 the project is summarised and potential future studies are identified.
2 Overview of sonar sensors

2.1 Sensing the environment using sonar

One of the most commonly used sensors for indoor robot navigation and localisation today is sonar (Sound Navigation and Ranging), the operating environment of many robotic programs suits the specifications of this type of sensor. Combined with the fact that they are cheap and easy to use, makes this an ideal choice for many applications.

Sonar is used to measure the distance between the robot and objects in its surroundings. The sonar transmits an ultrasonic signal (chirp) that is then reflected from the object (Figure 1). The reflected signal is then detected by the sonar. Generally, one chirp is fired and the same sensor listens for the echo. The software measures the time between the transmitted and received ultrasonic signal. Since we know the speed that the signal travelled (speed of sound), we can calculate a good estimation of the distance between the robot and object. The distance information can be processed in a number of ways to provide further information about the environment and the robot’s position within it.

![Ultrasonic Waves traveling at S_o](image)

*Figure 1. Time of flight (TOF) – sonar principle [1]*

Assuming that the speed \( S_0 = 340 \text{m/s} \) and that the time of flight (TOF) = \( \Delta T \), it is possible to calculate the distance (D) using the formulae below:

\[
\Delta T = \frac{2 \times D}{340}, \quad 1)
\]

\[
D = \frac{\Delta T \times 340}{2} \quad 2)
\]

However, the primary drawback with sonar is that the transmitted ultrasonic signal is cone-shaped with a certain beam width. This means that even though we can obtain a good estimate of the object’s distance, we cannot as easily determine the angle to it within the cone (Figure 2). This result in a lack of spatial resolution in the information obtained from the measurements.
There are various strategies for improving the accuracy of the bearing. A couple of strategies include utilising multiple sensors or the firing of the same sensor multiple times. Then by applying triangulation mathematics, spatial resolution can be improved. However, these strategies all require multiple firings, and since only one sensor can fire at a time, the processing time becomes prohibitive.
3 Original system

3.1 System specification

Research conducted at CAS using their sonar systems is generally focused on navigation, map building, positioning and collision avoidance. The original sonar systems were mounted on mobile robot platforms. In general, the sonar system consisted of sixteen Polaroid 6500 sonar sensors mounted in a ring formation in the x-y plane around the robot platform (Figure 3).

![Figure 3. a) The original mobile robot platform  b) Sonar sensors mounted in a ring formation](image)

The sensors are PC controlled operating with Linux. The original set-up allowed only one sensor to be used at a time and the system only detected the closest object that happened to be present within the cone at the time of measurement. The sonar array is generally used as a research system in a robot laboratory operated by trained researchers. The target environment is set up to mimic a typical human indoor environment.

3.2 Polaroid 6500 series sonar ranging module

Sensors of all kinds have their limitations and therefore require a specific operating domain. For economic reasons, it has been a design and construction strategy to use simple and cheap sensor components in robots. The Polaroid 6500 sonar sensor has been the sensor of choice of many researchers including those based at CAS.

The sonar transducers have limited range due to the energy loss of the ultrasound signal that decreases in strength with distance at an inverse squared proportion in air. The maximum range for an ultrasonic transducer in air is about 10 m. It is common that the sonar transducer is used for both sending and receiving the chirp. When the sonar is sending it works as a speaker and sends out the chirp. As with a speaker, the membrane
oscillates and it takes some time for the membrane to reset. The same membrane is used to detect the chirp, so it needs to stop oscillating before it can start listening for the echo. Therefore it is not possible to make measurements during this time, and thus there will be a minimum distance the sonar can detect. This distance is from 0.1 to 0.3 m. The operating range is typically 0.2-10 m for currently available sensors.

### 3.2.1 Control interface

There are three input signals and one output signal to control the sonar as shown in Figure 4:

- **INIT**: Controls when to start the measurement process, i.e. the sensor sends out a chirp. The INIT signal is set to high.

- **BINH**: This signal is set to high when the sensor should start to listen after the echo. The default blanking time (Internal Blanking) is 2.38 ms, and it will be used if BINH is set to low. BINH input can be taken high to end the blanking of the receive input anytime prior to the internal blanking.

- **BLNK**: Controls the multiple echoes feature. When an echo is being detected by the transducer the ECHO output signal change from low to high. To be able to detect another echo the ECHO output signal must be reset back to low. To reset the ECHO output the BLNK must be set to high and then back to low for at least 0.44 ms in duration, to account for all 16 returning pulses from the target and allow for internal delay times.

- **ECHO**: The output signal when the sensor detects an echo.
To be able to listen for echo, the sonar has to first send out a chirp. This makes it impossible to receive multiple echoes of the same transmission by different sensors. Thus only one sensor is used at a time in the original system.

### 3.3 Estimating position by triangulation

There are methodologies available for obtaining more accurate information in regards to the angle between the robot and the object. The most common method is to take a range measurement with one sensor, move the robot a short distance, and then make a new range measurement from the new position. The two range measurement data can then be used to calculate the current angle. In advanced systems, arrays of sonars are being used for listening to both their own and the other sensors’ echoes in order to find object positions through triangulation [2].

The measured distance basically provides, for each object and sensor, a circular arc centred at the sensor location and with a radius equal to the distance between the sensor and the object. A locus vector for each object can thereafter be calculated using basic triangulation (Figure 5).
The intersection between the arcs defined by the readings, specifies the location of the common target. This intersection can be found as the solution of the following system of equations:

\[(X_T - X_{s1})^2 + (Y_T - Y_{s1})^2 = r_1^2\]  
\[(X_T - X_{s2})^2 + (Y_T - Y_{s2})^2 = r_2^2\]

\[\arctan \left(\frac{Y_T - Y_{s1}}{X_T - X_{s1}}\right) \in [\gamma_1 - \delta/2, \gamma_1 + \delta/2]\]  
\[\arctan \left(\frac{Y_T - Y_{s2}}{X_T - X_{s2}}\right) \in [\gamma_2 - \delta/2, \gamma_2 + \delta/2]\]

Where \((X_s, Y_s)\) denotes the sensor position, \(r\) is the range reading, \(\gamma\) the sensor heading angle and \(\delta\) the opening angle of the center sonar lobe. The target position \(X_T, Y_T\) can be derived from above equations.

In the practical application of the triangulation theory, a hypothesis algorithm is used to get the target position (Figure 6). A 2-hypothesis can be created from three sonar measurements by taking the mean position from two triangulations \((P_1 + P_2)/2\).
The idea of fusing sonar measurements from the same target [2]

3.4 Weaknesses

The ultrasound propagates at the ‘speed of sound’, approximately 340 m/s. A robot to object distance of 10 m requires the sound to travel 20 m to and from the object. This means that the time it takes to do one measurement will be at least 0.0588 s (20 m / 340 m/s). To determine the position via triangulation you will need at least three measurements from the same object, preferably more for better accuracy. If the robot moves between measurements, position estimation will take several seconds. This time delay will make it difficult or even impossible to obtain accurate position information in complex environments, particularly where there are moving objects.

To sum up, the basic weakness of the original system is that it is limited to using one sensor at a time. The consequences of this technical limitation are:

- that the system is unable to provide angular information about detected objects in a single session, due to only once sensor can be used at a time;
- that the system is unable to provide data about multiple echoes, thus is only able to detect the closest object;
- that multiple measurements take a long time to obtain. This limits the ability to handle moving objects (low temporal resolution); and
- Limited accuracy in distance estimation due to problems with averaging over multiple sensor inputs.
4 Goals, requirements and limitations

4.1 General goal

The overall goal was to design and build a generic sensor platform that would enable a robot to detect objects in its environment. The platform would need to have improved accuracy compared to the equipment already in use at CAS. The platform should use Polaroid 6500 ultrasonic sensors. They were chosen for the project due to their popularity, ease of use and economic benefits. The aim was that the platform should be directly applicable in any robot application based on a standard PC.

4.2 Assumptions regarding the operating domain

4.2.1 Indoor environment

The sonar array is to be used as a research system in a robot laboratory. It is assumed that the target environment is a fairly normal indoor environment, and the system will be operated by trained researchers. The research on robot navigation is conducted using robot platforms. The system is therefore constructed accordingly to be used with the robot platform.

4.2.2 Choice of sensor

For economic reasons, it has been a design and construction strategy to use simple and cheap basic sensor components in robots. The weaknesses previously described all originate from features of the simple sensors used and the fact that only one sensor is used at a time in the original system. Given the customer demand to use the original sensors over any other type of sensor, the room for improving the system is to use multiple sensors at the same time.

4.3 Requirements

There were a number of requirements that needed to be fulfilled for the project to be considered a success:

- The electronic between Nomad Robot platform and sonar sensors had to be implemented.
  - The system were to be implemented on a micro-processor. The micro-processor should be able to control the sonar array, do time measurement and send the measurement results to a Linux based PC on the Nomad platform.
  - Communication interface between micro-processor and Nomad platform were to be implemented.
  - The electronics between the micro-processor and the sonar sensors that controls the sonar array were to be implemented.

- The user of the sensor platform had to be able to control the system and obtain resultant data through a C++ class interface.
- The system had to be reasonably expandable in order to suit future needs and facilitate changes in the host system.
5  Solution

5.1  Basic approach

The approach to the given task has been bottom-up, beginning with an examination of the basic components and specific features of the original system, then exploring possible strategies to enhance these features towards the goals of the work. One of the main priorities has been to provide flexibility, and thus suitability for the research purposes of the end product. In more technical terms, this has leaned towards modular expandability and the preservation of the low-level parameters and degrees of freedom given by the basic sensor hardware.

From an architectural viewpoint there has been a focus on compensating for the shortcomings of the sensors by integrating multiple sensors in a cooperative manner, facilitating efficient and straightforward sensor integration on a higher level (host level).

The basic solution was to enable sonar sensors to receive echoes without transmitting, and allowing multiple sensors to transmit at the same time. By doing so, multiple measurements can be obtained within a single session and the object position can be calculated.

5.2  Position estimation by multiple receptions

If an echo from the same transmission can be read by different sensors, the location of the target can be derived by using the same formulas as described in section 3.3, see Figure 7.

\[
(X_T - X_s)^2 + (Y_T - Y_s)^2 = r_1^2
\]

\[
(X_T - X_s)^2 + (Y_T - Y_s)^2 = r_2^2
\]

Where \((X_s, Y_s)\) denotes the sensor position.

\(r\) is the range reading where:

\[
r_1 = (T_{s1} - T_0) \times 340 / 2
\]

\[
r_1 + r_2 = (T_{s2} - T_0) \times 340 / 2
\]

The target position \(X_T, Y_T\) can be derived from above equations.
5.3 System overview and architecture

The overview of the system is shown in Figure 8. The sonar array is controlled by a Sensor Controller that is connected to a Linux based PC via RS232.

The sensor controller makes it possible to control all the sensors simultaneously, and takes advantage of the full functionality of the sensors. This is the most essential part in the system and is built around a microcontroller. The transmission and reception functionality of the sensors are also decoupled by the controller, so that sensors can receive echoes without transmission.

The C++ program running on a PC provides a user interface to control the whole system and retrieve data.
Figure 8. System overview
6 Hardware platform design

6.1 The Polaroid 6500 sonar sensor

The Polaroid 6500 sonar sensor is composed of two main parts: an acoustical transducer and a ranging module. The transducer is an electrostatic type that is used to transmit the signal and to receive the echo. Basically it works the same way as a speaker and a microphone. The ranging module contains all of the circuitry needed to generate the transmit signal, drive the transducer, receive the echo, and process the analog information received by the transducer.

The basic principle to get the distance information is to measure the time between the sending and return of the chirp. With the knowledge of the speed of sound it is then simple to calculate the distance. The ranging module doesn’t have an onboard time measurement unit so it has to be added externally.

The digital interface to control the sensor is described in section 0.

6.1.1 Operating requirements

Operating requirements:

- Voltage supply (Vcc) = 5 V (Maximum 7 V);
- Vcc Operating free-air temperature range 0º C - 40º C;
- Supply current during transmit period 2A, after transmission 100 mA;
- During transmit period, 16 pulses at 49.4 KHz are being transmitted; and
- After applying power a minimum of 5 ms must elapse before the measurement can start.

6.1.2 Performance and accuracy

The sensor can measure distances in the interval 0.2 m – 10 m. The typical absolute accuracy is ± 1% of the reading over the entire range.

6.2 Modifications of the sensor electronics

The sonar transducer original sensor control functionality does not allow a listening period without first sending out a chirp. To have multiple sensors listening for the transmission of one chirp, the original sensor control must be modified. By removing the connection between the transmission and reception circuitry, the sensor array can be configured so that multiple sensors can cooperate flexibly. Figure 9 shows the electrical schematic of Polaroid 6500.
There are a total of 9 pins of which 7 are occupied. The modification incorporates installing a relay between XMIT and Q1, using the free pins 3 and 5 (Figure 10). By doing so, transmission of a chirp by a sensor can be controlled by setting the SEL.
6.3 Sensor controller

The sensor controller hardware was built around a microcontroller. The key is to be able to control all the sensors at the same time. All sensors can then be configured to fire at the same time and receive echoes at the same time.

The implementation that was chosen uses 4 input signals and 1 output signal from each sonar sensor:

- **INIT** (IN): Set to 1 if the sonar is chosen to be active;
- **SEL** (IN): Set to 1 if the sonar is chosen to transmit out a chirp;
- **BINH** (IN): Set to 1 if the sonar is chosen to listen after echoes;
- **BLNK** (IN): Set to 1 after received echo for 0.44ms, then set back to 0, to reset ECHO output signal from 1 to 0 to allow the receiving of another echo; and
- **ECHO** (OUT): Set to 1 if the sonar has received echo. Set back to 0 by BLINK.

Given 16 sensors, there are 80 signals totally. The ordinary cheap off-the-shelf microcontrollers do not have 80 I/O ports that can be controlled at the same time. However, the 80 signals are not needed at the same time. With help of a multiplexer (MUX), it is possible to switch between different signals.

To choose between five different signals, at least three control pins are needed to code the address and select the signal to set:

- INIT: 000
- BLNK: 001
- BINH: 010
- ECHO: 011
- SEL: 100

After the signal is chosen, 16 data signals are used to set the value of the signal for 16 individual sensors.

The Sensor Controller is designed using a microchip, PIC 17C44/JW, connecting to a number of MUX boards. All MUX boards share the same address lines (three pins) connected to the Peripheral Interface Controller (PIC). Each MUX has four data lines that control the four sonar sensors it connects. See Figure 11 below.

In the case of controlling 16 sonar sensors, four MUX boards are needed, and a total of 16 data lines are needed from the PIC.
PIC specification

Microchip PIC 17C44/JW was used. It has:

- a built in EPROM with 8Kbyte of memory for the PIC control program;
- 454 bytes of data memory for storage of the distance measurements;
- 33 I/O pins with individual direction control;
- RS-232 communication interface to communicate with the PC;
- A 16-bit timer/counter with 8-bit programmable prescaler for time/distance measurement; and
- 32MHz clock input and 125 ns instruction cycle.

6.3.2 Mux specification

Each mux-card consists of one 74HCT238 3-to-8 line decoder and five 74HCT573 D-type latches, one D-latch for every signal to control (sel, init, binh, blnk and echo). All the D-latches are connected to the same data bus. The decoder is controlled by an address bus. The decoder activates only one D-latch at a time. Then data can be written to the D-latch through the data bus. When the D-latch is deactivated it will hold the data written to it. The data and address bus are controlled by the PIC. The sel, init, binh, blnk signals go to the transducers. The echo signal goes from the transducer.
Figure 12 shows the connection of MUX card to one sensor. Each MUX card is designed to receive 4 separate data input from the PIC, therefore can control four sensors. This is to be compliant to the requirement that the system should be expandable. For current system with 16 sonar sensors, 4 MUX cards needed. It’s a simple modification to add more MUX to the PIC to support more sensors.

All MUX card share the same 3 pins of address bus and adding one MUX card needs 4 pins of data bus. Give that max 33 I/O pins are available, max 7 MUX cards and 28 sensors can be used at the same time.

![Figure 12. MUX](image)

### 6.3.3 PC-PIC interface

There must be a way to communicate between the PC and the PIC. The PC must provide the PIC with configuration data. The PIC must return the measurement values to the PC. The easiest way to do this is to use the RS-232 data communication interface. It is one of the most used and stable communication interfaces. Mostly every PC has the RS-232 hardware implemented. The PC is running Linux which has support for the RS-232 interface and the chosen PIC has a built in RS-232 interface.
**Desired data rate: B9600**

The baud rate can be calculated from following formula:

\[
\text{Desired Baud rate} = \frac{\text{Fosc}}{64 (X + 1)}.
\]

Fosc is the PIC’s oscillator frequency 32 MHz.

X is the register SPBRG value, which specifies the PIC’s baud rate.

This mean there will always be an error in the baud rate. The baud rate 9600 is chosen because this gives the smallest error from the baud rate generated by the PIC. The X should be 51 for the baud rate 9600, which gives the error of 0.16%.

**Start stop control**

A simple transfer protocol has been implemented. The transfer of parameters from the PC to the PIC starts with a ‘b’ and ends with an ‘e’. The transfer of data from the PIC to the PC will continue until the PIC ends the transfer with three ‘0x00’ in a row.

See appendices 1 and 2 to view the MUX electronic schematics and the PIC electronic schematics respectively.
7 Software design

There are two pieces of SW running on the PC that provides a control interface for the user to setup the configuration parameters for the sonar array. The configuration parameters are sent to PIC board via RS-232. Another piece of SW running on the PIC board receives configuration parameters, sets corresponding D-latches, receives echoes and sends the echo time and sonar numbers back to the PC.

See Figure 13 for an overview picture of the Sonar array control SW.
Initialize SonarArray()
Open Serial Port
Set time step = 1us
Set meas time = 0.1s
Set blanking time = 2.38ms
Set echo reset time = 1ms
Clear all sonar buffer

setActiveSonar(i)
setFireSonar(i)
measure()

write data to serial port
fire SONAR
measureTime
activeSonar
blankingTime
echoResetTime

init all 5 Dlatch
set SELDlatch according to fireSonar
set INIT Dlatch according to activeSonar
start timer
wait for blankingTime
set BINH Dlatch of activeSonar

While (time < measureTime) {
  save received time
  save sonar number n
  set BLNK of echoSonar
  for echoResetTime
}

timeMeasure
SonarNumber

Save Data

return getDistanceMeasure()

Distance Calculation

return

print out

Figure 13. Sonar array control SW overview
7.1 PC software

The PC software is a control interface to the measurement system written in C++.

Sonar_array.cc & sonar_array.h

These two files provide a set of C++ classes that:

- Set/clean control parameters for 16 sonar sensors;
- Setup RS-232 communication;
- Transmit and receive data via RS-232;
- Save received echoes and sonar sensor numbers; and
- Calculation of the measurement distance;

See Appendices 3 and 4 for the source code.

Sonar_array_main.cc

The main program is written by the user to initialise the sonar array, set configuration parameters and get the measurement result. All are done by calling the functions provided by sonar_array.cc/h. Depending on the different tasks, the main program is written differently. An example of a main program is given in Appendix 5.

7.2 Sonar array controller

There is a sonar array control program running on the PIC board. The main purpose of the controller is to:

- Get the control parameters from the PC;
- Control the sonar sensors via D-latch according to the control parameters;
- Record the time measurement on the received echo signals; and
- Send back the measurement date to the PC.

Control parameters received from the PC:

- active_sonar: 16 bits to set the INIT of 16 sonar sensors. This parameter controls which sensors are active. The sensors which are active can transmit chirp and/or receive echo;
- firing_sonar: 16 bits to set the SEL of 16 sonar sensors. This parameter controls which sensor to transmit chirp;
- measure_time: set the total measurement time of active sonar sensors;
- blanking_time: set BINH of active sensors after blanking time. All active sensors shall listen to the echo; and
- echo_reset_time: set BLNK of sonars that received echo for echo reset time.
Figure 14 shows the flow chart of the sonar array control program that runs on the PIC board. See Appendix 6 for the source code.
8 Conclusion and future work

In this thesis I show the development and implementation of a complete system that supports flexible configuration of a sonar array:

- A Sonar Controller including a PIC and 4 MUX have been implemented between the Nomad Robot platform and sonar sensors;
- RS-232 was used as interface between the Sonar Controller and a Linux-based PC on the Nomad platform; and
- A C++ program was developed so that a user can control the sensors and obtain results through a C++ class interface.

The system supports an array of 16 sonar sensors and can be easily expended to max 28 sensors. It enables the usage of multiple sensors at the same time and thus can reduce the sonar measurement time dramatically.

In chapter 2 a simple algorithm is described, that uses multiple echoes to determine the target position. Olle Wijk, a Ph.D student at CAS, used the sonar controller built for three-dimensional sonar scans (unpublished).

Figure 15 below shows a sonar array device mounted on top of a Superscout. Each sonar plays the role of four sensors by rotating the robot in 90 degree 4 steps. Between each rotation all sensors are fired. Hence, 64 readings distributed as plot in Figure 16 become available from each robot position.

Figure 15. Sonar array for 3D scan
Figure 16. Sonar beams for 3D sonar scan

The work by Olle Wijk outlined above illustrates a potential use of the modified sonar array platform. To conclude, the flexibility of configuring and controlling the sonar array provided by the Sonar Controller opens the door for new sonar measurement algorithms that can increase the accuracy of target positions, obtain 3D positions, and decrease measurement time.
9 References


Appendix 1: Mux Card
Appendix 2: PIC Card

PIC Card Oscillator
PIC Card RS232
PIC Card Power
Appendix 3: Sonar_array.h

#ifndef _INCLUDE_sonar_array_H
#define _INCLUDE_sonar_array_H

class sonar_array
{
  public:

    /* creates class sonar_array whit device argument for rs232 communication */
    sonar_array(char *dev);

    ~sonar_array();

    /* Functions for activating sonars */

    /* Activates a specific sonar */
    void set_active_sonar(int sonar_nr);
    /* Activates all sonars */
    void set_all_active_sonar();
    /* Deactivates a specific sonar */
    void clear_active_sonar(int sonar_nr);
    /* Deactivates all sonars */
    void clear_all_active_sonar();
    /* Check if a specific sonar is activated */
    bool is_active_sonar(int sonar_nr);

    /* Functions for firing of sonars */

    /* Sets a specific sonar to fire */
    void set_firing_sonar(int sonar_nr);
    /* Disable a specific sonar to fire */
    void clear_firing_sonar(int sonar_nr);
    /* Disables all sonars to fire */
    void clear_all_firing_sonar();
    /* Check if a specific sonar is set to fire */
    bool is_firing_sonar(int sonar_nr);

    /* Functions for timing control */

    /* Sets blanking time in seconds */
    void set_blanking_time (float time_s);
    /* Gets blanking time in seconds */
    float get_blanking_time ();

    /* Sets time in seconds between multiple echoes */
    void set_echo_reset_time (float time_s);
    /* Gets time in seconds between multiple echoes */
    float get_echo_reset_time ();}
/* Sets total measure time in seconds */
void set_measure_time(float time_s);
/* Gets total measure time in seconds */
float get_measure_time();

/* Sets sound speed in meter per second */
void set_sound_speed(float sound_speed);
/* Gets sound speed in meter per second */
float get_sound_speed();

/* Sets time in seconds for a raw sonar data timestep */
void set_time_step(float time_s);
/* Gets time in seconds for a raw sonar data timestep */
float get_time_step();

/* Sets time offset in seconds */
void set_time_offset(float time_offset);
/* Gets time offset in seconds */
float get_time_offset();

/* Sets distance offset in meters */
void set_distance_offset(float distance_offset);
/* Gets distance offset in meters */
float get_distance_offset();

/* executes measurement */
void measure();

/* Functions for measurement handling */

/* Get number of echos for a specific sonar */
int get_no_of_echo(int sonar_nr);

/* Gets measurement in raw sonar data from a specific sonar echo*/
float get_raw_measure(int sonar_nr, int echo_nr);
/* Gets measurement in seconds from a specific sonar echo */
float get_time_measure(int sonar_nr, int echo_nr);
/* Gets measurement in meters from a specific sonar echo */
float get_distance_measure(int sonar_nr, int echo_nr);

private:

/* Functions for device handling and communication */

/* Opens a specified device for communication */
void open_port(char *dev);
/* Deletes device buffer */
void flushport();
/* Variable for device status */
int serial_fd;

/* Writes control data to sonars */
void write_control_data();
/* Reads raw sonar measurement from sonars */
void read_sonar_data();

/* Variables for sonar control */

/* Start char for sonar communication */
unsigned char start_char;

/* Hi char for controlling activation of sonars */
unsigned char active_sonar_hichar;
/* Lo char for controlling activation of sonars */
unsigned char active_sonar_lochar;

/* Hi char for controlling firing of sonars */
unsigned char firing_sonar_hichar;
/* Lo char for controlling firing of sonars */
unsigned char firing_sonar_lochar;

/* Hi char for controlling activation of sonars */
unsigned char measure_time_hichar;
/* Lo char for controlling activation of sonars */
unsigned char measure_time_lochar;

/* Hi char for controlling blanking time for sonars */
unsigned char blanking_time_hichar;
/* Lo char for controlling blanking time for sonars */
unsigned char blanking_time_lochar;

/* Hi char for controlling time between multiple echoes for sonars */
unsigned char echo_reset_time_hichar;
/* Char for controlling time between multiple echoes for sonars */
unsigned char echo_reset_time_lochar;

/* End char for sonar communication */
unsigned char end_char;

/* Vector for sonar control data */
unsigned char outbuf[14];

/* Variables sonar data handling */

/* Vector for receiving data from sonar */
unsigned char pic_indata[1000];
/* Matrix for raw sonar data */
unsigned int sonar_raw_measure[16][300];
/* Vector for number of echoes from each sonar */
int no_of_echo[16];
/* Variable for raw sonar data time step in seconds */
float time_step;
/* Variable for time offset in seconds */
float time_offset;
/* Variable for sound speed in meter per second */
float sound_speed;
/* Variable for distance_offset in meters */
float distance_offset;

/* help functions */

/* Converts sonar data to int */
unsigned int char_to_int(unsigned char hichar, unsigned char lochar);
/* Converts sonar data to float */
float char_to_float(unsigned char hichar, unsigned char lochar);
/* Converts int to sonar data */
void int_to_char(unsigned int intvalue, unsigned char& hichar, unsigned char& lochar);
/* Converts float to sonar data */
void float_to_char(unsigned int floatvalue, unsigned char& hichar, unsigned char& lochar);

/* Sets a specified bit in sonar data */
void char_bit_set(int bitno, unsigned char& hichar, unsigned char& lochar);
/* Sets all bits in sonar data */
void char_all_bit_set(unsigned char& hichar, unsigned char& lochar);
/* Clears a specified bit in sonar data */
void char_bit_clear(int bitno, unsigned char& hichar, unsigned char& lochar);
/* Clears all bits in sonar data */
void char_all_bit_clear(unsigned char& hichar, unsigned char& lochar);
/* Tests if a bit is set in sonar data */
bool char_bit_test(int bitno, unsigned char& hichar, unsigned char& lochar);

};
#endif
Appendix 4: Sonar_array.cc

#include "sonar_array.h"
#include <stdio.h> /* Standard input/output definitions */
#include <string.h> /* String function definitions */
#include <unistd.h> /* UNIX standard function definitions */
#include <fcntl.h> /* File control definitions */
#include <errno.h> /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <iostream.h>
#include <sys/time.h>
#include <sys/select.h>

#define bitset(var,bitno) ((var)|=1<<(bitno))
#define bitclear(var,bitno) ((var)&=~(1<<(bitno)))
#define bittest(var,bitno)  (((var)&(1<<(bitno)))!=0)

#define BAUDRATE B9600
#define _POSIX_SOURCE 1 /* POSIX compliant source */
#define FALSE 0
#define TRUE 1
#define default_blanking_time 2.38e-3 /* Minimum 2.38 ms */
#define default_echo_reset_time 1e-3 /* Minimum 0.44 ms */
#define default_time_step 1e-6 /* (1/(32MHz/4))*8 */
#define default_time_offset 0
#define default_sound_speed 340
#define default_distance_offset 0
#define default_measure_time 0.1

/* creates class sonar_array whit device argument for rs232 communication */
sonar_array::sonar_array(char *dev)
{
  open_port(dev);

  /* default sonar control data */
  set_time_step( default_time_step );
  clear_all_active_sonar();
  clear_all_firing_sonar();
  set_measure_time ( default_measure_time );
  set_blanking_time ( default_blanking_time );
set_echo_reset_time ( default_echo_reset_time );
set_time_offset( default_time_offset  );
set_sound_speed( default_sound_speed  );
set_distance_offset ( default_distance_offset );

start_char='b';
end_char='e';
}

sonar_array::~sonar_array()
{
  close(serial_fd);
}

/* Functions for activating sonars */

  /* Activates a specific sonar */
  void sonar_array::set_active_sonar(int sonar_nr)
  {
    char_bit_set((sonar_nr), active_sonar_hichar, active_sonar_lochar);
  }

  /* Activates all sonars */
  void sonar_array::set_all_active_sonar()
  {
    char_all_bit_set(active_sonar_hichar, active_sonar_lochar);
  }

  /* Deactivates a specific sonar */
  void sonar_array::clear_active_sonar(int sonar_nr)
  {
    char_bit_clear((sonar_nr), active_sonar_hichar, active_sonar_lochar);
  }

  /* Deactivates all sonars */
  void sonar_array::clear_all_active_sonar()
  {
    char_all_bit_clear(active_sonar_hichar, active_sonar_lochar);
  }

  /* Check if a specific sonar is activated */
  bool sonar_array::is_active_sonar(int sonar_nr)
  {
    return char_bit_test((sonar_nr), active_sonar_hichar, active_sonar_lochar);
  }
/* Functions for firing of sonars */
/* Sets a specific sonar to fire */
void sonar_array::set_firing_sonar(int sonar_nr)
{
    char_bit_set((sonar_nr), firing_sonar_hichar,
                 firing_sonar_lochar);
}

/* Disable a specific sonar to fire */
void sonar_array::clear_firing_sonar(int sonar_nr)
{
    char_all_bit_clear(firing_sonar_hichar, firing_sonar_lochar);
}

/* Disables all sonars to fire */
void sonar_array::clear_all_firing_sonar()
{
    char_all_bit_clear(firing_sonar_hichar, firing_sonar_lochar);
}

/* Check if a specific sonar is set to fire */
bool sonar_array::is_firing_sonar(int sonar_nr)
{
    return char_bit_test((sonar_nr), firing_sonar_hichar,
                         firing_sonar_lochar);
}

/* Functions for timing control */
/* Sets blanking time in seconds */
void sonar_array::set_blanking_time (float time_s)
{
    unsigned int charvalue;
    charvalue=(unsigned int)(time_s/time_step);
    int_to_char(charvalue, blanking_time_hichar,
                blanking_time_lochar);
}

/* Gets blanking time in seconds */
float sonar_array::get_blanking_time ()
{
    return (time_step*char_to_float(blanking_time_hichar,
                                    blanking_time_lochar));
}

/* Sets time in seconds between multiple echos */
void sonar_array::set_echo_reset_time (float time_s)
{
    unsigned int charvalue;
    charvalue=(unsigned int)(time_s/time_step);
    int_to_char(charvalue, echo_reset_time_hichar,
                echo_reset_time_lochar);
}
/* Gets time in seconds between multiple echos */
float sonar_array::get_echo_reset_time()
{
    return (time_step*char_to_float(echo_reset_time_hichar,
    echo_reset_time_lochar));
}

/* Sets total measure time in seconds */
void sonar_array::set_measure_time(float time_s)
{
    unsigned int charvalue;
    charvalue=(unsigned int)(time_s/time_step);
    int_to_char(charvalue, measure_time_hichar, measure_time_lochar);
}

/* Gets sound speed in meter per second */
float sonar_array::get_measure_time()
{
    return (time_step*char_to_float(measure_time_hichar,
    measure_time_lochar));
}

/* Sets time in seconds for a raw sonar data timestep */
void sonar_array::set_time_step(float time_s)
{
    time_step = time_s;
}

/* Gets time in seconds for a raw sonar data timestep */
float sonar_array::get_time_step()
{
    return time_step;
}

/* Sets time offset in seconds */
void sonar_array::set_time_offset(float time_s)
{
    time_offset = time_s;
}

/* Sets sound speed in meter per second */
void sonar_array::set_sound_speed(float speed_m_s)
{
    sound_speed = speed_m_s;
}

/* Gets time offset in seconds */
float sonar_array::get_time_offset()
{
    return time_offset;
}

/* Gets sound speed in meter per second */
float sonar_array::get_sound_speed()
{
    return sound_speed;
}

/* Sets distance offset in meters */
void sonar_array::set_distance_offset(float distance_m)
{
    distance_offset = distance_m;
}

/* Gets distance offset in meters */
float sonar_array::get_distance_offset()
{
    return distance_offset;
}

/* Executes measurement */
void sonar_array::measure()
{
    int i;
    int k;
    char temp;

    for (i=0;i<=15;i++)
    {
        no_of_echo[i]=0;
        for (k=0;k<=299;k++)
            sonar_raw_measure[i][k]=0;
    }
    //flushport();
    write_control_data();
    read_sonar_data();
}

/* Functions for measurement handling */

/* Get number of echos for a specific sonar */
int sonar_array::get_no_of_echo(int sonar_nr)
{
    return no_of_echo[sonar_nr];
}

/* Gets measurement in raw sonar data from a specific sonar echo*/
float sonar_array::get_raw_measure(int sonar_nr, int echo_nr)
{
    return (float)sonar_raw_measure[sonar_nr][echo_nr];
}

/* Gets measurement in seconds from a specific sonar echo */
float sonar_array::get_time_measure(int sonar_nr, int echo_nr)
{ 
    return (float) ( (get_raw_measure(sonar_nr, echo_nr) * 
    time_step) + time_offset); 
}

/* Gets measurement in meters from a specific sonar echo */
float sonar_array::get_distance_measure(int sonar_nr, int 
    echo_nr) 
{ 
    return (((get_time_measure(sonar_nr, echo_nr)/2) * sound_speed) 
    + distance_offset); 
}

/* Opens a specific device for communication */
void sonar_array::open_port(char *dev) 
{
    struct termios oldtio, newtio;

    /* open port */
    serial_fd = open(dev, O_RDWR | O_NOCTTY);
    if (serial_fd < 0)
        cout << "read fail on port " << dev << endl;
    else
        cout << "port open " << endl;

    /* save current port settings */
    tcgetattr(serial_fd, &newtio);

    /* write zeros to newtio */
    bzero(&newtio, sizeof(newtio));

    /* sets newtio to raw stream */
    cfmakeraw(&newtio);

    /* Set the baud rates */
    cfsetispeed(&newtio, BAUDRATE);
    cfsetospeed(&newtio, BAUDRATE);

    newtio.c_cc[VMIN] = 0;
    newtio.c_cc[VTIME] = 10;

    /* clears port */
    tcflush(serial_fd, TCIFLUSH);

    /* read returns immediately */
    // fcntl (serial_fd, F_SETFL, FNDELAY);

    /* activates new settings */
    tcsetattr(serial_fd, TCSANOW, &newtio);
void sonar_array::flushport()
{
    struct timeval tv;
    fd_set rfd;
    int nbytes = 0;
    char buf;

    /*
     * Clear the set of read file descriptors, and
     * add the two we just got from the open calls.
     */
    FD_ZERO( &rfd );
    FD_SET( serial_fd, &rfd );

    /*
     * Set a 1 second timeout.
     */
    tv.tv_sec = 1;
    tv.tv_usec = 0;

    int ready = 0;
    while(!ready){
        switch ( select( 1 + serial_fd, &rfd, 0, 0, &tv ) ) {
            case -1:
                perror( "sonar_array(flushport):select" );
                exit( EXIT_FAILURE );
            case 0:
                ready = 1;
                break;
            default:
                if( FD_ISSET( serial_fd, &rfd ) )
                    nbytes += read(serial_fd, &buf, 1);
        }
    }
    cout << "nbytes = " << nbytes << endl;
}

/* Writes control data to sonars */
void sonar_array::write_control_data()
{
    int send=10;
    outbuf[0]=start_char;
    outbuf[1]=active_sonar_hichar;
    outbuf[2]=active_sonar_lochar;
    outbuf[3]=firing_sonar_hichar;
    outbuf[4]=firing_sonar_lochar;
    outbuf[5]=measure_time_hichar;
    outbuf[6]=measure_time_lochar;
    outbuf[7]=blanking_time_hichar;
outbuf[8]=blanking_time_lochar;
outbuf[9]=echo_reset_time_hichar;
outbuf[10]=echo_reset_time_lochar;

write(serial_fd,&start_char,1);
write(serial_fd,&active_sonar_hichar,1);
write(serial_fd,&active_sonar_lochar,1);
write(serial_fd,&firing_sonar_hichar,1);
write(serial_fd,&firing_sonar_lochar,1);
write(serial_fd,&measure_time_hichar,1);
write(serial_fd,&measure_time_lochar,1);
write(serial_fd,&blanking_time_hichar,1);
write(serial_fd,&blanking_time_lochar,1);
write(serial_fd,&echo_reset_time_hichar,1);
write(serial_fd,&echo_reset_time_lochar,1);
write(serial_fd,&end_char,1);

//if (write(serial_fd,&outbuf, send)<send)
//  cout<< "send error" << endl;
// cout << "write done"<< endl;
}

/* Reads raw sonar measurement from sonars */
void sonar_array::read_sonar_data()
{
    int n = 999;
    struct timeval tv;
    fd_set rfd;
    int nbytes = 0;

    int k;
    unsigned char raw_measure_hi;
    unsigned char raw_measure_lo;
    unsigned char sonar_nr_hi;
    unsigned char sonar_nr_lo;
    int i;
    int idata;

    /*
    * Clear the set of read file descriptors, and
    * add the two we just got from the open calls.
    */
    FD_ZERO( &rfd );

    */
FD_SET( serial_fd, &rfd );

/*
 *    Set a 2 second timeout.
 */
tv.tv_sec = 2;
tv.tv_usec = 0;

cout << "begin read" << endl;

while((nbytes < n) && (nbytes != -1) &&
((pic_indata[nbytes-1] + pic_indata[nbytes-2] +
  pic_indata[nbytes-3]) != 0)){
    switch ( select( 1 + serial_fd, &rfd, 0, 0, &tv ) ) {
        case -1:
            perror( "sonar_array:select" );
            exit( EXIT_FAILURE );
        case 0:
            perror( "sonar_array:select timed out" );
            nbytes = -1;
            break;
        default:
            if( FD_ISSET( serial_fd, &rfd ) )
                nbytes += read(serial_fd,
                &pic_indata[nbytes], n);
            // cout << "nbytes = " << nbytes << endl;
    }
}

cout << "nbytes = " << nbytes << endl;

for (idata=0; (idata < (nbytes-4)); idata+= 4) {
    // cout<<"idata = "<< idata << endl;
    raw_measure_hi = pic_indata[idata];
    raw_measure_lo = pic_indata[idata+1];
    sonar_nr_hi = pic_indata[idata+2];
    sonar_nr_lo = pic_indata[idata+3];

    for (k=0;k<=7;k++)
    {
        /* put measure that corresponds to sonar nr in sonar_data_raw */

        if ((sonar_nr_hi&(1<<k))!=0)
            no_of_echo[k+8] ++;

    }
sonar_raw_measure[k+8][(no_of_echo[k+8])]=(raw_measure_hi << 8)
| raw_measure_lo;

}

if ((sonar_nr_lo&(1<<k))!=0)
{
    no_of_echo[k] ++;
    sonar_raw_measure[k][(no_of_echo[k])]=(raw_measure_hi
<< 8 ) | raw_measure_lo;
}

}

for (i=nbytes-4;i<nbytes;i++)
cout << " pic_data "<<(int)pic_indata[i]<<(endl;
cout << " char to transmit left = "<<(int)pic_indata[nbytes-
4]<< endl;
}

/* help functions */

/* Converts sonar data to int */
unsigned int sonar_array::char_to_int(unsigned char hichar, unsigned char lochar)
{
    return ((hichar << 8 ) | lochar);
}

/* Converts sonar data to float */
float sonar_array::char_to_float(unsigned char hichar, unsigned char lochar)
{
    return ((hichar << 8 ) | lochar);
}

/* Converts int to sonar data */
void sonar_array::int_to_char(unsigned int intvalue, unsigned char& hichar, unsigned char& lochar)
{
    lochar=(unsigned char)intvalue;
    hichar=(unsigned char)(intvalue>>8);
}

/* Converts float to sonar data */
void sonar_array::float_to_char(unsigned int floatvalue, unsigned char& hichar, unsigned char& lochar)
{
    int_to_char( (int)floatvalue, hichar, lochar);
}

/* Sets a specified bit in sonar data */
void sonar_array::char_bit_set(int bitno, unsigned char& hichar, unsigned char& lochar)
{
    if (bitno<8)
        bitset(lochar,(bitno));
    else
        bitset (hichar,(bitno-8));
}

    /* Sets all bits in sonar data */
void sonar_array::char_all_bit_set(unsigned char& hichar, unsigned char& lochar)
{
    hichar=0xFF;
    lochar=0xFF;
}

    /* Clears a specified bit in sonar data */
void sonar_array::char_bit_clear(int bitno, unsigned char& hichar, unsigned char& lochar)
{
    if (bitno<8)
        bitclear(lochar,(bitno));
    else
        bitclear(hichar,(bitno-8));
}

    /* Clears all bits in sonar data */
void sonar_array::char_all_bit_clear(unsigned char& hichar, unsigned char& lochar)
{
    hichar=0;
    lochar=0;
}

    /* Tests if a bit is set in sonar data */
bool sonar_array::char_bit_test(int bitno, unsigned char& hichar, unsigned char& lochar)
{
    if (bitno<8)
        return bittest(lochar,(bitno));
    else
        return bittest(hichar,(bitno-8));
}
Appendix 5: Sonar_array_main.cc

#include <iostream.h>
#include "sonar_array.h"
#define MODEMDEVICE "/dev/cua1"

void main()
{
 int i;
 int k;
 int sonarnr;
 int svar;

 sonar_array sonar_array(MODEMDEVICE);

 for(;;)
 {
   // cin >> svar;
   for (i=0;i<=15;i++)
   {
     sonar_array.set_active_sonar(i);
     sonar_array.set_firing_sonar(i);
   }

   sonar_array.measure();
   sonar_array.clear_all_firing_sonar();

   sonar_array.set_firing_sonar(8);
   sonar_array.measure();

   for (k=0;k<=15;k++)
   {
     cout<< " sonar nr"<< k << " = " <<
     sonar_array.get_no_of_echo(k)<< "; ";
     for (i=1;i<=sonar_array.get_no_of_echo(k);i++)
     {
       cout<< sonar_array.get_distance_measure(k,i)<<"; ";
     }
     cout<<endl;
   }
 }
}
Appendix 6: Sonar.c

/* continuous serial transfer, Real clock, real ports, clear echo, reset measurement that is not used, mask unused sonars, unsigned char */
/* multiple echos */

#include <pic.h>
#include <stdio.h>
#include <stdlib.h>

#define bitset(var,bitno) ((var)|=1 << (bitno))
#define bitclr(var,bitno) ((var)&=~(1<<((bitno))))
#define bittst(var,bitno) (((var)&(1<<(bitno)))!=0)

#define baudrate 9600
#define ClkFreq 32000000
#define baudSPBRG ((10*ClkFreq/(64*baudrate))+5)/10 - 1
#define vSPBRG 52 /* baudrate 9600 till SPBRG */
#define TXSTA_INIT 0x20 /* 8-bit transmission, async mode */
#define RCSTA_INIT 0x90 /* 8-bit reception, enable serial port, enable reception */

#define TRUE 1
#define FALSE 0

#define vec_length 40

/* Writes lo and hi byte to D-latch on adres portEadr */

write_Dlatch (unsigned char portEadr, unsigned char hichar, unsigned char lochar) {
    PORTE=0xFF; /* disconnect all D-latches */

    PORTE=portEadr; /* Adres Dlatch on adres Eadr */
    PORTE=0xFF; /* disconnect all D-latches */

    PORTC=0x00; /* clear portc before setting direction */
    DDRC=0x00; /* set portc to output */

    PORTD=0x00; /* clear portd before setting direction */
    DDRD=0x00; /* set portd to output */

    PORTC=lochar; /* writes lo byte to portc */
    PORTD=hichar; /* writes hi byte to portd */

    PORTC=0x00; /* clear portc */
    PORTD=0x00; /* clear portd */
    PORTC=FALSE; /* disable portc */
    PORTD=FALSE; /* disable portd */

    write_Dlatch (portEadr, hichar, lochar);
}

/* reads and returns a char from serial port */
unsigned char get_serial_data_poll ()

/* multiple echos */

#include <pic.h>
#include <stdio.h>
#include <stdlib.h>

#define bitset(var,bitno) ((var)|=1 << (bitno))
#define bitclr(var,bitno) ((var)&=~(1<<((bitno))))
#define bittst(var,bitno) (((var)&(1<<(bitno)))!=0)
{  
    while (bittst(PIR,0) == 0);  
    return RCREG;
}

/* writes a char to serial port */
void send_serial_data_poll (unsigned char o)  
{  
    while (bittst(PIR,1) == 0);
    TXREG=o;
}

main()
{
    /*variables*/
    /* count varaiable */
    unsigned char isave; /* index for next save in vector */
    unsigned char idata; /* index for which data to send */
    unsigned char isend; /* send index in vector */
    unsigned char iecho; /* index for next echo to activate in vector */

    /* sonar config data from serial port */
    unsigned char startbit='b';
    unsigned char active_sonar_hichar, active_sonar_lochar; /* control active sonars */
    unsigned char firing_sonar_hichar, firing_sonar_lochar; /* witch sonar that is going to be fired* /
    unsigned char measure_time_hichar, measure_time_lochar; /* measurement duration */
    unsigned char blanking_time_hichar, blanking_time_lochar; /* interrupts internal blanking if < 2,38ms*/
    unsigned char echo_reset_time_hichar, echo_reset_time_lochar; /* Blanking time for multiple echos*/
    unsigned char endbit='e';

    /* Sonar data */
    unsigned char time_measure_hichar[vec_length],
    time_measure_lochar[vec_length]; /* time measure from sonar */
    unsigned char sonar_nr_hichar[vec_length],
    sonar_nr_lochar[vec_length]; /* sonar number for time measure*/

    /* BLNK buffert for resetting echo */
    unsigned char blanking_portc;
    unsigned char blanking_portd;
/ * variables for multipple echo */
unsigned char blanking_time_calc; /* is set true if blanking
time ned to be calculated, false otherwise */
unsigned char echo_active_time_hichar;
unsigned char echo_active_time_lochar;

/* test variable for transmission */
unsigned char char_left;

/* initiating port E to outport, d-latches turnd off*/
PORTE=0x00;
DDRE=0x00;
PORTE=0x07;

/* initiating USART for rs-232 communication */
SPBRG=vSPBRG;
TXSTA=TXSTA_INIT;
RCSTA=RCSTA_INIT;

for (;;) {

/* resets measurment values
isend=0;
do{
  time_measure_hichar[isend]=0x00;
  time_measure_lochar[isend]=0x00;
  sonar_nr_hichar[isend]=0x00;
  sonar_nr_lochar[isend]=0x00;
  isend ++;
}while(isend<16);

/* Geting sonar config data from serial port */
while (get_serial_data_poll() != startbit);

active_sonar_hichar = get_serial_data_poll();
active_sonar_lochar = get_serial_data_poll();

firing_sonar_hichar = get_serial_data_poll();
firing_sonar_lochar = get_serial_data_poll();

measure_time_hichar = get_serial_data_poll();
measure_time_lochar = get_serial_data_poll();

blanking_time_hichar = get_serial_data_poll();
blanking_time_lochar = get_serial_data_poll();

echo_reset_time_hichar = get_serial_data_poll();
echo_reset_time_lochar = get_serial_data_poll();

while (get_serial_data_poll() != endbit);
/* init timer0, reset clock, prescaler to 1:8*, clock source
to external (no signal) */

bitset(CPUSTA, GLINTD); /* disable all interrupt */
T0STA = 0x06; /* set clocksource to tock1 (no signal),
prescaler to 1:8*/
TMR0L=0x00; /* Reset timer0 lo */
TMR0H=0x00; /* Reset timer0 hi */
T0STA = 0x06; /* set clocksource to tock1, prescaler to
1:8*/
bitclr(CPUSTA, GLINTD); /* enable all interrupt */

/* Init sonars */
write_Dlatch (0x00, 0x00, 0x00); /* clear init Dlatch */
write_Dlatch (0x01, 0x00, 0x00); /* clear Blnk Dlatch */
write_Dlatch (0x02, 0x00, 0x00); /* clear Binh Dlatch */
write_Dlatch (0x04, firing_sonar_hichar, 
firing_sonar_lochar); /* sets sonar fire to sel Dlatch */

/* init variables */
isave=0x00;
idata=0x00;
isend=0x00;
iecho=0x00;
blanking_time_calc=TRUE;
blanking_portc=0x00;
blanking_portd=0x00;
echo_active_time_hichar = 0x00;
echo_active_time_lochar = 0x00;

/* fire sonars */
write_Dlatch (0x00, active_sonar_hichar, 
active_sonar_lochar); /* set init Dlatch hi */

/* start_clock() */
bitset(CPUSTA, GLINTD); /* disable all interrupt */
bitset(T0STA, 5); /*sets Timer0 clock sourse to internal clock */
bitclr(CPUSTA, GLINTD); /* enable all interrupt */

/* Wait BINH blanking time */
while ((TMR0H<blanking_time_hichar) ||
(TMR0L<blanking_time_lochar));
write_Dlatch (0x02, active_sonar_hichar, 
active_sonar_lochar); /* End internal blanking */

/* init echo */
PORTE=0xFF; /* disconnect all D-latches */
PORTC=0x00; /* clear portc befor setting direction */
DDRC=0xFF; /* set portc to Input */

PORTD=0x00; /* clear portd before setting direction */
DDRD=0xFF; /* set portd to Input */

PORTE=0x03; /* Mux to echo D-latch */

/* polling for sonar echoes and save them */
do {
    /* if port changes save values */
    if (((PORTD & active_sonar_hichar) != 0) || ((PORTC & active_sonar_lochar) != 0))
    {
        time_measure_lochar[isave] = TMR0L;
        time_measure_hichar[isave] = TMR0H;
        sonar_nr_lochar[isave] = PORTC & active_sonar_lochar;
        sonar_nr_hichar[isave] = PORTD & active_sonar_hichar;
        blanking_portc = (blanking_portc | sonar_nr_lochar[isave]); /* remember echo bits on portc */
        blanking_portd = (blanking_portd | sonar_nr_hichar[isave]); /* remember echo bits on portd */

        /* clear echo */
        PORTE=0xFF; /* disconnect all D-latches */

        PORTC=0x00; /* clear portc before setting direction */
        DDRC=0x00; /* set portc to output */

        PORTD=0x00; /* clear portd before setting direction */
        DDRD=0x00; /* set portd to output */

        PORTC=blanking_portc; /* writes low byte to portc */
        PORTD=blanking_portd; /* writes high byte to portd */

        PORTE=0x01; /* Address BLNK D latch */

        PORTE=0xFFF; /* disconnect all D-latches */

        /* set D-latch to echo */
        PORTC=0x00; /* clear portc before setting direction */
        DDRC=0x0FF; /* set portc to Input */

        PORTD=0x00; /* clear portd before setting direction */
        DDRD=0x0FF; /* set portd to Input */

        PORTE=0x03; /* Mux to echo D-latch */

        /* Check if end of vector */
        if (isave==(vec_length-1))
{  
  if (isend!=0)  
    isave = 0;  
}

else if ((isave + 1) != isend)  
  isave++;  
}

if (iecho != isave)  
{  
  if (blanking_time_calc==TRUE)  
  {  
    if ( (measure_time_hichar - time_measure_hichar[iecho]) < echo_reset_time_hichar )  
    {  
      echo_active_time_hichar = 0xFF;  
      echo_active_time_lochar = 0xFF;  
    }  

    if ( ((0xFF - time_measure_lochar[iecho]) < echo_reset_time_lochar ) && ( (measure_time_hichar - time_measure_hichar[iecho]) > echo_reset_time_hichar ) )  
    {  
      echo_active_time_lochar = time_measure_lochar[iecho] + echo_reset_time_lochar;  
      echo_active_time_hichar = time_measure_hichar[iecho] + echo_reset_time_hichar + 1;  
    }  

    if ( ((0xFF - time_measure_lochar[iecho]) > echo_reset_time_lochar ) && ( (measure_time_hichar - time_measure_hichar[iecho]) > echo_reset_time_hichar ) )  
    {  
      echo_active_time_lochar = time_measure_lochar[iecho] + echo_reset_time_lochar;  
      echo_active_time_hichar = time_measure_hichar[iecho] + echo_reset_time_hichar;  
    }  

    blanking_time_calc=FALSE;  
  }  

  if ((TMR0H >= echo_active_time_hichar) && (TMR0L >= echo_active_time_lochar) )  
  {  
    blanking_portc = (blanking_portc & (sonar_nr_lochar[iecho]^0xFF));  /* reset echo bits on portc */  
    blanking_portd = (blanking_portd & (sonar_nr_hichar[iecho]^0xFF));  /* reset echo bits on portd */  
  }  
}
/ clear echo */
PORTE=0xFF; /* disconect all D-latches */

PORTE=0xFF; /* disconect all D-latches */
PORTC=0x00; /* clear portc befor seting direction */
DDRC=0x00; /* set portc to output */

PORTD=0x00; /* clear portd befor seting direction */
DDRD=0x00; /* set portd to output */

PORTC=blanking_portc; /* writes lo byte to portc */
PORTD=blanking_portd; /* writes hi byte to portd */

PORTE=0x01; /* Adres BLNK Dlatch */

PORTE=0xFF; /* disconect all D-latches */

/* set D-latch to echo */
PORTE=0xFF; /* disconect all D-latches */
PORTC=0x00; /* clear portc befor seting direction */
DDRC=0xFF; /* set portc to Input */

PORTD=0x00; /* clear portd befor seting direction */
DDRD=0xFF; /* set portd to Input */

PORTE=0x03; /* Mux to echo D-latch */

blanking_time_calc=TRUE;

if (iecho==(vec_length-1))
    iecho = 0;
else
    iecho++;
}

/* if port changes save values */
if (((PORTD & active_sonar_hichar) != 0) || ((PORTC & active_sonar_lochar) != 0))
{

time_measure_lochar[isave] = TMR0L;
time_measure_hichar[isave] = TMR0H;
sonar_nr_lochar[isave] = PORTC & active_sonar_lochar;
sonar_nr_hichar[isave] = PORTD & active_sonar_hichar;
blanking_portc = (blanking_portc |
sonar_nr_lochar[isave]); /* remember echo bits on portc */
blanking_portd = (blanking_portd |
sonar_nr_hichar[isave]); /* remember echo bits on portd */

/* clear echo */

PORTE=0xFF; /* disconect all D-latches */
PORTC=0x00; /* clear portc before setting direction */
DDRC=0x00;  /* set portc to output */

PORTD=0x00; /* clear portd before setting direction */
DDRD=0x00;  /* set portd to output */

PORTC=blanking_portc;  /* writes lo byte to portc */
PORTD=blanking_portd;  /* writes hi byte to portd */

PORTE=0x01; /* Address BLNK D latch */

PORTE=0xFF; /* disconnect all D-latches */

/* set D-latch to echo */
PORTC=0x00; /* clear portc before setting direction */
DDRC=0x00;  /* set portc to input */

PORTD=0x00; /* clear portd before setting direction */
DDRD=0xFF;  /* set portd to input */

PORTE=0x03; /* Mux to echo D-latch */

/* Check if end of vector */
if (isave==(vec_length-1))
  {
    if (isend!=0)
      isave = 0;
  }
else if ((isave+1) != isend)
  isave++;

if (isend != isave)
  {
    if (bittst(PIR,1) != 0)
    {
      switch (idata){
      case 0x00:
        TXREG=time_measure_hichar[isend];
        idata=0x01;
        break;
      case 0x01:
        TXREG=time_measure_lochar[isend];
        idata=0x02;
        break;
      case 0x02:
        TXREG=sonar_nr_hichar[isend];
        idata=0x03;
        break;
      case 0x03:

TXREG=sonar_nr_lochar[isend];
idata=0x00;
if (isend==(vec_length-1))
   isend = 0;
else
   isend++;
break;
}

while((TMR0H < measure_time_hichar) || (TMR0L <
   measure_time_lochar));

char_left= isave - isend;
while (isend != isave)
{
   if (bittst(PIR,1) != 0)
   {
      switch (idata){
      case 0x00:
         TXREG=time_measure_hichar[isend];
         idata=0x01;
         break;
      case 0x01:
         TXREG=time_measure_lochar[isend];
         idata=0x02;
         break;
      case 0x02:
         TXREG=sonar_nr_hichar[isend];
         idata=0x03;
         break;
      case 0x03:
         TXREG=sonar_nr_lochar[isend];
         idata=0x00;
         if (isend==(vec_length-1))
            isend = 0;
         else
            isend++;
         break;
      }
   }
   send_serial_data_poll (char_left);
   send_serial_data_poll (0x00);
   send_serial_data_poll (0x00);
   send_serial_data_poll (0x00);
   send_serial_data_poll (0x00);