NTNU – Trondheim
Norwegian University of
Science and Technology

James the PokerBot

Part2: Playing Texas Hold'em

Malin Edvardsen

Master of Science in Engineering Cybernetics
Submission date: June 2012
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

James- The PokerBot

Part2: Playing Texas Hold'em

This report contains information on all the components needed to create the first prototype of James the PokerBot. This prototype is created to play a real life game of Texas Hold'em, against human players, using a normal deck of cards.

First Texas Hold'em then doing the dishes and finally world dominance!

Malin Edvardsen

6/7/2012

Assignment

Card games are an enjoyable activity. But without someone to play with the possibilities are reduced. This is often the situation when staying at a hospital or while waiting for the plane at an airport. Especially the elderly may not have the same joy and opportunity by playing with for example a PC. A cheap card-playing robot could then be used for this, not least as a demonstration tool for motivating students in relation to further studies.

The task will consist of:

- To the extent necessary, get familiarized with the background theory and assess previous work on the subject
- Suggest possible solutions and design
- To the extent time permits, implementing a unified solution
- Assess the suitability of the obtained solution

Summary

James the PokerBot is a robot created to play a game of Texas Hold'em against human players using a normal deck of cards.

The idea of making James came from a project in a computer vision course. The goal for this project was creating a computer vision program that would identify the playing card in a picture of a single card. As the project proved very interesting, it was decided to continue the work as a separate project after the computer vision course ended. The resulting program would be able to identify several playing cards from a picture, despite there being other non-card objects in this image. More about this previous project can be read in the report that was made for it, fall 2011 [1] that is included in the digital attachment.

This report however, contains a description of the steps needed to go from having the computer vision program to make the beginning of a card playing robot.

The first part of the report deals with the changes that needed to be done to the old computer vision program. These changes includes getting input from a live source rather than using static images, as well as speed optimization needed when running the program on a less powerful BeagleBoard. A video showing the raw output of the final program is included in the digital attachment under *Result videos, Computer vision* as well as being available online [2].

Next the report continues by describing the different modules used to play the game. First the main module is explained. This corresponds to the actual robot and is the module that runs the computer vision program as well as controls the progression of the game itself. The programs made to run on the main module were developed on a stationary computer. For the prototype made in this project, a development board called a BeagleBoard would work as the main module.

The other modules made, were the player modules. These were small controllers used by the human players to interface with James when playing the game. Three different player modules have been made. First a simulator was made to use when developing the programs for the main module. Then two different hardware prototypes, both based on AVR microcontrollers, were made for the final setup.

The report then describes the way communication was setup between the modules. For the simulator player modules, this was achieved using virtual COM ports. The hardware prototypes on the other hand, used XBee modules communicating wirelessly over ZigBee.

Following this is a chapter describing how the actual Texas Hold'em application was designed. This application controls the game structure and handles input from the human players regarding the game. The application will also interpret the raw input from the computer vision program, regarding the current state of the game.

Finally there is a chapter describing the robotic arm planned to use for picking up playing cards. Sadly there was never enough time to complete this arm, but ideas on how this would be done are discussed. There are also added pictures of an unfinished prototype that was made.

Two videos demonstrating the final results both for the Texas Hold'em application running on the stationary computer, as well as for the hardware prototypes, are added in the digital attachment under *Result videos* as well as being available online [3] [4].

The first video demonstrates how the computer vision program can be used for a machine to effectively play a game of cards against human players. The second video demonstrates how these programs also can be used with relatively inexpensive hardware, making it possible to commercialize the product.

1.1 Acknowledgements

I would like to thank the Department of Engineering Cybernetics at NTNU, for allowing me to have this project as my master thesis the spring 2012.

I would also like to thank Amund Skavhaug for having been my supervisor during the project as well as Richard E. Blake for his help during the initial computer vision project in the course *TDT4265 Computer Vision*, spring 2011.

Sammendrag (Norwegian)

James the PokerBot er en robot laget for å spille kortspillet Texas Hold'em mot menneskelige motspillere ved å benytte en helt vanlig kortstokk.

Ideen om å lage James kom fra et prosjekt i et datasyn kurs ved NTNU. Målet for dette prosjektet var å lage et datasyn program som skulle identifisere spillkort i et bilde av ett enkelt kort. Ettersom prosjektet vist seg svært interessant, ble det besluttet å videreføre arbeidet som et eget prosjekt etter at datasyn kurset var avsluttet. Det resulterende programmet ville være i stand til å identifisere flere spillkort i et bilde, til tross for andre objekter i bildet som ikke var spillekort. Mer om prosjektet kan leses i rapporten [1] som er inkludert i det digitale vedlegget.

Denne rapporten inneholder en beskrivelse av fremgangsmåten som ble brukt for å gå fra datasyns programmet til å lage en kort spillende robot.

Den første delen av rapporten omhandler de endringer som måtte gjøres for det gamle datasyns programmet. Disse endringene omfatter å få input fra et webkamera istedenfor å bruke statiske bilder, samt hastighets optimalisering som var nødvendig for å kjører programmet på et BeagleBoard. En video som viser direkte output fra det endelige programmet er inkludert i det digitale vedlegget under *Result videos*, *Computer vision* samt å være tilgjengelig på nettet [2].

Videre fortsetter rapporten med å beskrive de ulike modulene som brukes for å spille spillet. Først er hovedmodulen forklart. Denne tilsvarer den faktiske roboten og er modulen som kjører datasyns programmet samt styrer utviklingen av selve spillet. Programmene laget for hovedmodulen ble utviklet på en stasjonær pc. For prototypen laget i dette prosjektet, ble et utviklings brett kallet et BeagleBoard benyttet som hoved modul.

De andre modulene brukt, var spiller modulene. Dette er små kontrollere som benyttes av de menneskelige spillere som grensesnitt til James når man spiller spillet. Tre forskjellige spillere moduler har blitt laget. Først en simulator som ble brukt under utvikling av programmene for hovedmodulen. Deretter ble to forskjellige fysiske prototyper, begge basert på AVR mikrokontrollere, laget for det endelige oppsettet.

Deretter beskriver rapporten hvordan kommunikasjon ble satt opp mellom modulene. For den simulerte spilleren modulene, ble dette oppnådd ved hjelp av virtuelle COM-porter. Prototypene derimot, brukte XBee moduler som kommuniserer trådløst ved hjelp av ZigBee.

Neste kapittel beskriver hvordan selve Texas Hold'em applikasjonen ble utformet. Dette programmet styrer spillets struktur samt håndterer innspill fra menneskelige spillere. Denne applikasjonen vil også tolke input fra datasyns programmet om den nåværende tilstanden i spillet.

Til sist er det et kapittel som beskriver robotarmen som var planlagt å lage for å plukke opp spillkort. Dessverre var det aldri nok tid til å fullføre denne armen, men ideer om hvordan dette ville bli gjort er diskutert. Det er også lagt med bilder av en uferdig prototype som ble laget.

To videoer som viser de endelige resultatene både for Texas Hold'em applikasjon som kjører på den stasjonære maskinen, samt den endelige prototypen for James the pokerBot, er lagt til i det digitale vedlegget under *Result videos*, samt at de er tilgjengelige på nettet [3] [4].

Den første videoen viser hvordan datasyn programmet kan brukes for at en maskin skal kunne effektivt spille et kortspill mot menneskelige motspillere. Den andre videoen viser hvordan disse programmene også kan brukes med relativt billig maskinvare, noe som gjør det mulig å kommersialisere produktet.

1.2 Takk til

Jeg vil takke instituttet for teknisk kybernetikk ved NTNU, for å tillate meg å ha dette prosjektet som min masteroppgave våren 2012.

Jeg vil også takke Amund Skavhaug for å ha vært min veileder i løpet av prosjektet, samt Richard E. Blake for hans hjelp ved det innledende datasyn prosjektet i TDT4265 Computer Vision, våren 2011.

Contents

James- The PokerBot

Part2: Playing Texas Hold'em

2 Introduction

Have you ever wanted to play a game of real life poker, but had no friends to play with?
Perhaps you actually do have a friend or two, but would prefer one more player at your table.
Or maybe you just like cool gadgets!

Well look no further! James the PokerBot is here!

2.1 James the PokerBot

James had been an idea, a long time before the making of James the PokerBot. The thought was that you would have James as a base robot and then could buy different hardware sets to open up for different software packages. It would be an open system, so that independent companies could make different hardware and software pieces, and you as a customer could by the parts you wanted to build up you own personal robot.

An example would be the card game set, where you would be able to download support for different card games, as well as make (and sell) your own software to work with the given hardware set. Another example would be a company selling dishwashers, making a patch so that the "clean the table and put plates and cups in the dishwasher" set for James would be compatible to the given dishwasher.

The idea of James the PokerBot came from a project where it was attempted to make a playing card recognizing program. The project was not too successful but very interesting, and it was decided to continue work on it. When the program became usable, it was decided to attempt creating a robot that would use it to play a game of cards. The choice of game fell on Texas Hold'em as you then only have two cards on your hand. This means it would need a robotic arm to pick up the cards, without the task being too complex.

This project was first of all aiming to make a working system, not a perfect system. The robot would be able to use the computer vision program as input for a game of Texas Hold'em. It would then play the game against an undefined number of human players.

2.2 Previous work

When working on this project, no documentation was found on previous attempts to build a poker playing robot. Still, several of the parts of this project have been achieved individually. This includes computer vision programs made to detect playing cards [4] [5] [6] [7], programs used to play online poker [8], Texas Hold'em applications [9] and dealer robots [10] [11] [12] There are also several projects using point-to-multipoint wireless communication, as well as projects running on BeagleBoards and/or AVR based modules. Still, most help solving the challenges encountered was found reading different online forums on the subjects.

A Poker Bot, or Poker Robot, will normally be associated with a piece of software used to play online poker. It makes decisions on the current hand being played and acts accordingly. Some of the most popular poker bots are listed and reviewed in a page made by purely-poker.com [8]. Algorithms used to analyze poker hands relative to the game situation, could be used to improve James's poker skills.

2.3 Total overview of the project

The following figure is meant to give a total overview of all the components made to create James the PokerBot.

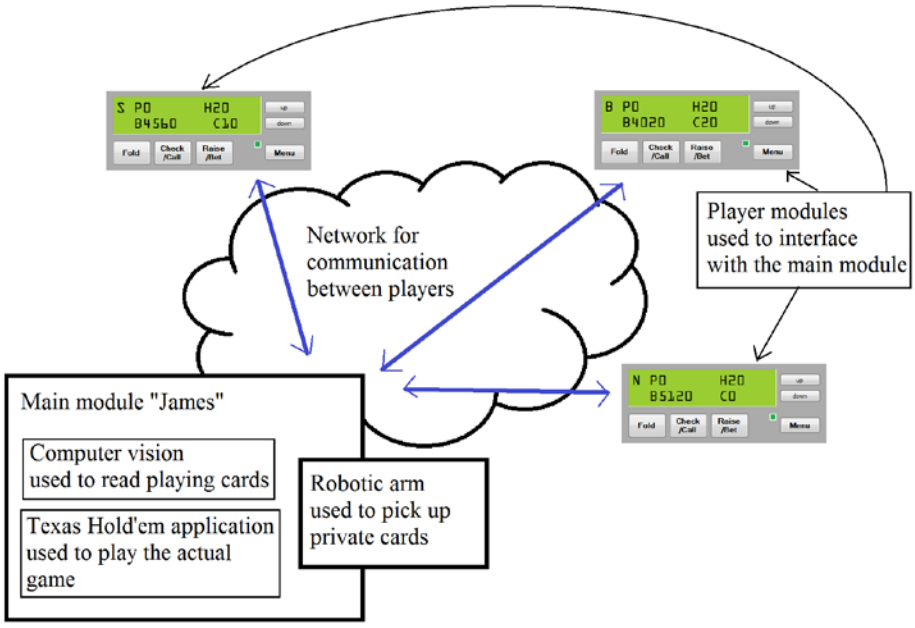


Figure 2-1: Overview

Form the figure, it can be seen how playing a game with James the PokerBot, will work sort of like playing a game of real life internet poker. As James do not (yet) understand human speech or is able to read poker chips, human players will need to use custom made player modules. The human players will then use these modules like they would use the buttons on an online poker site. Communication between modules is achieved using the wireless standard ZigBee.

The main module is connected to a camera, and use computer vision to identify playing cards. These cards come from a normal card deck, and the computer vision program can be configured to work with different decks. The Texas Hold'em application interprets output from the computer vision program to the active state of the game. This application is also what makes James able to actually play the game itself.

Finally a robotic arm was meant to be used by James to pick up "his" private pocket cards. Unfortunately there was not enough time to complete this part of the project.

2.4 Disposition of the report

This report has a rather orally presentation to ease the reading as it contains quite a large number of elements. For the same reason, subchapters will present and discuss part results where it is appropriate. The report is presented chronologically compared to how work was divided when working on the project. The main parts of the report contain the following;

2.4.1 Computer vision (Chapter 3)

The computer vision program used is mostly based on a previous project [1]. This program would return an array of cards detected from a static picture. Several improvements were done to the program, including live input, improving speed, auto-generating templates, improved corner voting and fixing of some known bugs.

2.4.2 Main Unit (Chapter 4)

The main module is used to run the computer vision program and the Texas Hold'em app used by James the PokerBot. The main module was first designed on a stationary computer, before being cross compiled to a BeagleBoard. The BeagleBoard was connected to, a cameras for the computer vision, an XBee for communication with the other players, a robotic arm for fetching pocket cards and a screen that was used for debugging. This screen also worked as a USB hub for connecting the camera, a keyboard, a mouse, a USB-Ethernet adapter and power for one of the player modules.

2.4.3 Player Modules (Chapter 5)

To communicate with James, all human players will need a player module. Three types of player modules were made. The first was a simulator made as a windows form application while the other two were physical prototypes based on AVR processors. A module would generally consist of a small display, 6 buttons and a communication device.

2.4.4 Communication (Chapter 6)

To communicate with the main module, the simulator program used virtual COM port pairs while the prototypes used XBee modules to communicate over ZigBee.

2.4.5 Playing Poker (Chapter 7)

The Texas Hold'em application made for this project follows the game rules added in appendix 13.1. The application takes raw data received from the computer vision program and interprets this information regarding the current state of the game. Actions taken by human players are received by the main unit from different player modules, while actions taken by the James unit are automatically generated within the application.

2.4.6 Robotic arm (Chapter 7)

As there was not enough time to complete the robotic arm, this chapter is made to discuss some of the ideas on how this could have been done.

2.4.7 Ending (Chapter 9, 10 and 11)

Finally, the main results from the project are presented and discussed before a final conclusion is made.

3 Computer Vision

The most challenging and time consuming task when making James the PokerBot was making the computer vision program. As James would be playing poker, using real life playing cards, it was necessary being able to detect and identify all cards on a table.

The program used was mostly based on a previous project [1]. In this old project, an incomplete playing card detector was made. It would detect cards as long as at least one of the cards corners was visible, independent on other object in the picture. Input to the program was provided as static photos from the computer running the program. This picture would then be processed through a series of different algorithms, resulting in an array of cards detected. The information could then be sent as a string of raw data to whatever application in need of the information.

Cards were detected using the following steps:

- Removing objects inside cards before finding lines in the picture
- Identify the lines by using a Hough diagram
- Locate card corners from the detected lines
- See if the card is facing downwards
- If not, then extract the rank and suite images from the detected corners and scaling them to template size
- Use correlation to identify the templates
- Finally, a voting algorithm would decide how a card would be identified if more than one corner was detected.
- The program would be able to detect cards in a picture, even if there were other unidentified objects in the camera view.

Several improvements were done to the program. This included getting live input, improving speed, auto-generating templates, improved corner voting and fixing of some known bugs from the previous version.

The raw data from the computer vision program would next be interpreted corresponding to the status of the game. As this would depend on the game being played, it was done in the Texas Hold'em application, and is further described in chapter 7.

The program was written in C++ using Visual Studio as IDE. It was developed on a computer using windows as OS. Later the program was cross compiled to run on a BeagleBoard having Angstrom (Linux for embedded devices) as OS (chapter 4.2).

3.1 Getting live input

As mentioned above, the previously made card recognizing program was not quite complete when starting this project. The most crucial improvement that needed to be done was getting input directly from a live source compared to using static pictures. This was achieved using OpenCV [13] to get frames directly from a web camera.

```
// Start capturing frames from camera using autodetect
(CV_CAP_ANY)
camCapture = cvCaptureFromCAM( CV_CAP_ANY );

// Get one frame from the camera
currentFrame = cvQueryFrame( camCapture );

// Allocate the grayscale image
currentFrameGrey = cvCreateImage(
cvSize( currentFrame->width, currentFrame->height ),
currentFrame->depth, 1 );

// Convert it to grayscale
cvCvtColor( currentFrame, currentFrameGrey, CV_RGB2GRAY );

// Get inputframe as float array for easyer access
for( y = 0; y < WINDOW_HEIGHT; y++ )
      for( x = 0; x < WINDOW_WIDTH; x++ ) {
            pos = x + y*WINDOW_WIDTH;
            inputFrame[pos] = ((pixel) currentFrameGrey-
>imageData[pos]);
      }
```

The digital attachment to this report contains a video demonstrating the computer vision program using live input. The video can also be found online [2].

3.2 Speed optimization

Another important improvement done to the program, was regarding speed. When running on a 3GHz computer, the processing time per frame was around 0.4-1.2 se. When the program was moved to a 720MHz BeagleBoard (chapter 4.2), this time increased to around 8-9 seconds when trying to identify 5 cards. This made the program quite unusable, and therefore it needed to be fixed.

There exist several performance tools that could be used when improving code like this. These tools include the gperftools [14] and the profiling tool for Visual Studio [15]. Using tools like this is generally recommended, as they will also provide information on things like memory usage, usage of particular instructions and frequency of function calls.

As the main goal for this project was making a complete system, not an ideal one, it was not prioritized spending time learning how to use these programs and how to adapt them to this system. Instead a simpler approach was used to measure improvement. This included using the `clock()` function from the `ctime` library. The function returns the number of clock ticks elapsed since the program was launched. By using it before and after critical functions and then looking at the difference between these two, it was possible see how much this function affected the total speed of the program. Following is an example of how this was done:

```
// Extract edges from the input frame
t1 = clock();
edgeSegmentation( );
t2 = clock();
diff = (float)(t2 - t1) / CLOCKS_PER_SEC;
cout << "  edgeSegmentation(): " << diff << endl;
```

As the BeagleBoard did not have an FPU, one of the biggest improvements was achieved by storing pixel values as unsigned 8 bit integer values instead of using float values. Two other important improvements were reducing cache misses by reordering for loops and pre calculating sine and cosine values used. These as well as other smaller changes resulted in the program going back to having a processing time on around 1-2 second per frame.

The pixels in the frames processed by the computer vision program were stored as grey values. When using float, the pixels would be represented as a value between 0 and 1. When using unsigned 8 bit integers on the other hand, the pixels would be represented as values between 0 and 255. This would affect a lot of the functions used, and therefore proved to be quite a big change to the program.

3.3 Auto template generation

Templates are small images that are used to identify detected cards through correlation. The correlation procedure compares the found rank and suit image to the templates. Then it returns the rank and suit that gave the best match.

Before auto generating, the templates were created manually. This was done by taking a picture of a card, cutting out the rank and suit images and sizing them into the desired template sizes. The templates would then be tested, and small changes were manually done to each template to make the correlation procedure more correct.

Figure 3-1: Manually generated templates

There were two reasons for implementing auto generation of templates. First it would make James the PokerBot independent of what card deck was used and second it was assumed to improve the correlation procedure.

To generate the templates, you would first decide what template you want to generate, and then place three cards with the wanted rank/suit on the table. The computer vision program would then run as usual until all four of the wanted symbols (rank or suit) for each card had been located. Then the template would be made as the average of each detected symbol and finally stored in the database folder.

```
// Generate template as average of each detected rank image
for( i = 0; i < RANK_WIDTH * TEMPLATE_HEIGHT; i++ )
      tempImg[i] = 0;
for( i = 0; i < cardCornersFound; i++ )
      addRankImg( i, tempImg );
for( i = 0; i < RANK_WIDTH * TEMPLATE_HEIGHT; i++ )
      tempImg[i] /= cardCornersFound;
storeTempImgAsTemplate( type, tempImg );
```

For the test deck, this resulted in the following templates.

Figure 3-2: Automatically generated templates

These templates were made from cards with the ID symbol in each card corner, but should also be possible to use with cards where only two of the corners contains the symbol. Then the number of cards added to the table should be increased from three to four, so that the amount of detected symbols would be at least eight (compared to twelve).

Also the expected size of the corner symbols should be found and stored when doing this, at that as well may vary from card deck to card deck.

3.4 Improved rank localization

The locating of rank and suit pictures in the corner of a card was one of the aspects in the program that needed improving. Two of the biggest problems here were regarding picture cards (J, Q and K) and heart of 4, 5, 6, 7, 8, 9 and 10.

For the picture cards, the problem appeared if the picture wasn't properly removed when removing corner background. This would result in the symbol not always being properly resized before correlation. Below the ranks detected from a king and a queen before and after improving localization has been added.

Figure 3-3: Before improved localization

Figure 3-4: After improved localization

We can here see how the ranks in Figure 3-4 are more evenly resized than those in Figure 3-3. This would then improve correlation for the picture cards. The improvement was done by searching the detected rank image for horizontal lines in the image that did not lie on top of any object pixels. The finally returned rank image would then contain the largest area of vertical lines lying on top of an object. The code that was used is added in appendix 13.2.1.1, as well as in the `locateValueAndSuit` function in the digital attachment under *BeagleBoard - Main unit*: *card.cpp*.

For the hearts, the problem was that the big heart would merge with the small heart, and therefore the small heart would be removed as background. The problem is illustrated below for the five, six and seven of hearts. Each with one heart detected as symbol (black) and one heart detected as background (white).

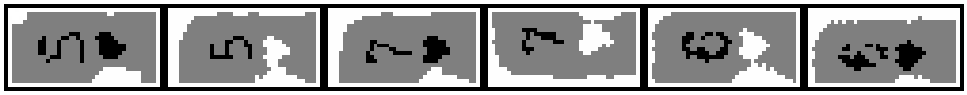


Figure 3-5: Dissapearing hearts

The problem with the hearts was never fixed as there was no time to do so. The idea was that when finding the width of the symbols, only the rank symbol would be considered. Then no pixels within this width should be considered background. As the rank never gets too close to any of the larger suit symbols on the card (at least not for this deck), that should have taken care of the problem.

3.5 Bugs fixed

Some of the minor bugs in the computer vision program were fixed during this project. These bugs included the merging cards bug, the disappearing vertical/horizontal edges bug and some memory leaks.

3.5.1 Merging cards

One of the problems in the old version of the computer vision program occurred when deciding the card ID from several corner IDs. This was done so that every time a corner was identified, the center point of that card was estimated. When having identified all card corners, a voting procedure was performed between all corners that had approximately the same card center point. This worked fine as long as cards were not overlapping. The problem occurred when overlapping cards had there center points at about the same position. The voting procedure would then not be able to separate the cards and therefore only return one final ID for both cards.

This problem was fixed by not only testing if the center points were located at the same place, but also if the corners actually could belong to the same card. If this was the case, the corner lines were tested to see if they were either parallel or orthogonal to each other, which would indicate that the corners in fact did belong to the same card. The code that was used is added in appendix 13.2.1.2, as well as in the `Card::sameCardAs` function in the digital attachment under *BeagleBoard - Main unit*: *card.cpp*.

3.5.2 Disappearing vertical/horizontal edges

When a card was put vertically or horizontally on the table relative to the camera, the edges would often partially disappear. The reason for this had root in how the variance of the regions in the image were calculated.

In the old version, when having calculated the variance of a region, the next calculation would step forward the space of an entire region. What would happen then was that an edge could fall between the regions, and therefore not be detected. An illustration of this is attempted in Figure 3-1. This dispays a possible card corner. The card is the light grey area and the backgournd the darker grey area. Regions were variance is calculated are marked as red squares and the once where the variance would be large enough to detect an edge marked as green. As seen, the top edge would not be detected, as the edge itsself falls between the regions.

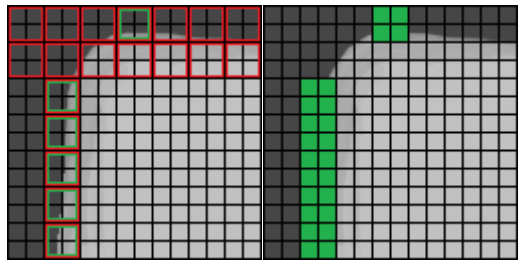


Figure 3-6: Dissapering edge

To fix this problem, variance was instead calculated by moving the region of interest only one pixel at a time. If the variance of the region was above a given threshold, the edge would be marked as the central pixel of that region, instead of marking the entire region. In Figure 3-7, the result of doing this for the same example as above is illustrated. The orange squares added here are the regions of interest that were not tested by doing this the old way. We can now see how the horizontal edge will be detected.

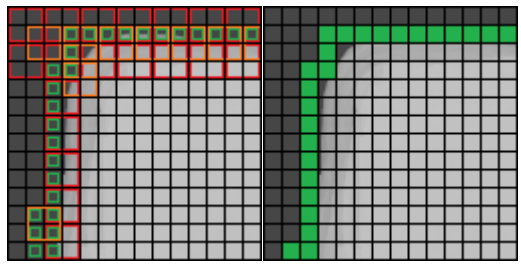


Figure 3-7: Edge detected

3.5.3 Memory leak

The program had a bad memory leak from when OpenCV was used to get the gray scale of the newly fetched frame. What happened was that the following function would allocate memory for a new image that then needed to be released

```
// Allocate the grayscale image
currentFrameGrey = cvCreateImage(
      cvSize( currentFrame->width, currentFrame->height ),
      currentFrame->depth, 1 );
```

Releasing the image was done using the following function

```
cvReleaseImage( &currentFrameGrey );
```

3.6 Results

A video shoving the raw result of the computer vision program getting live input is added in the digital attachment as well as published on youtube.com [2]. Some screenshots from the video are added further down.

The following two columns show the processing time of one frame before and after speed optimization, when running on the BeagleBoard. All results were generated running the computer vision program on the beagle board, at the same time, with the same camera settings and the same cards in the camera view.

```
Processing time, old version, 1 card
================================
resetCardRec( ): 1.14
playingCardRecognizing( ):
  edgeSegmentation(): 2.01
  identifyLines(): 0.97
  locateCorners(): 0.02
  getCardsFromCorners(): 0.1
  cardVoting( ): 0
playingCardRecognizing tot: 3.4
display image: 0.13
--------------------------------
- Total time per frame: 4.68
--------------------------------
```

```
Processing time, new version, 1 card
================================
resetCardRec( ): 0.23
playingCardRecognizing( ):
  edgeSegmentation(): 0.45
  identifyLines(): 0.07
  locateCorners(): 0
  getCardsFromCorners(): 0.03
  cardVoting( ): 0
playingCardRecognizing tot: 0.57
display image: 0.08
--------------------------------
- Total time per frame: 0.88
--------------------------------
```

```
Processing time, old version, 4 cards
================================
resetCardRec( ): 1.24

playingCardRecognizing( ):
  edgeSegmentation(): 1.97
  identifyLines(): 3.33
  locateCorners(): 0.69
  getCardsFromCorners(): 0.71
  cardVoting( ): 0.01
playingCardRecognizing tot: 7.01
display image: 0.11
--------------------------------
- Total time per frame: 8.36
--------------------------------
```

```
Processing time, new version, 4 cards
================================
resetCardRec( ): 0.23

playingCardRecognizing( ):
  edgeSegmentation(): 0.51
  identifyLines(): 0.14
  locateCorners(): 0.02
  getCardsFromCorners(): 0.22
  cardVoting( ): 0
playingCardRecognizing tot: 0.91
display image: 0.1
--------------------------------
- Total time per frame: 1.24
--------------------------------
```

```
Processing time, old version, 8 cards
================================
resetCardRec( ): 1.27
playingCardRecognizing( ):
  edgeSegmentation(): 1.99
  identifyLines(): 10.33
  locateCorners(): 2.06
  getCardsFromCorners(): 1.76
  cardVoting( ): 0
playingCardRecognizing tot: 16.45
display image: 0.13
--------------------------------
- Total time per frame: 17.85
--------------------------------
```

```
Processing time, new version, 8 cards
================================
resetCardRec( ): 0.23
playingCardRecognizing( ):
  edgeSegmentation(): 0.53
  identifyLines(): 0.25
  locateCorners(): 0.06
  getCardsFromCorners(): 0.37
  cardVoting( ): 0
playingCardRecognizing tot: 1.21
display image: 0.08
--------------------------------
- Total time per frame: 1.53
--------------------------------
```

Chart 1 illustrates the "Total time" for each of the examples above.

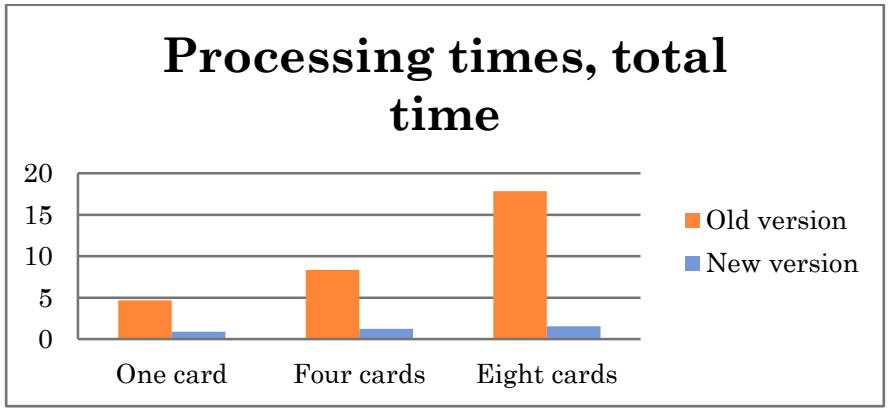


Chart 1: Processing times, total time

The following screenshots are from the video of the computer vision program that is added in the digital attachment.

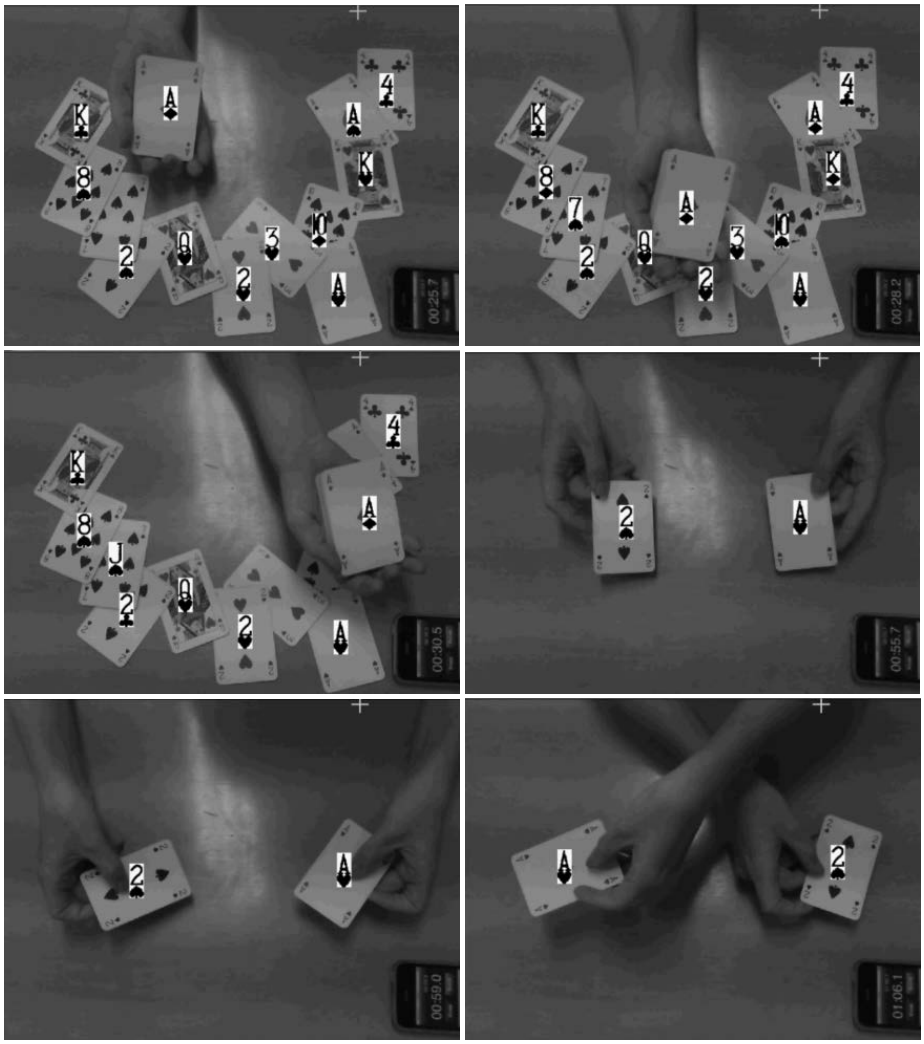


Figure 3-8: Screenshots from computervision program using live input

Chart 2 displays the percentage of wrongly identified ranks after correlation using manually and automatically generated templates. The data is achieved by running correlation on 300 corners of each rank, and registering each time the rank was wrongly identified. The final bar shows the average for all ranks using manually and automatically generated templates.

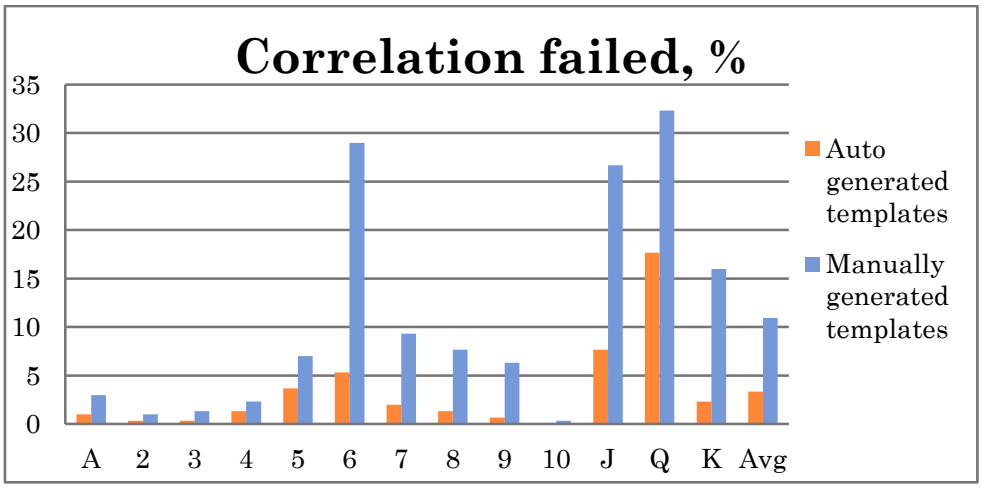


Chart 2: Correlation failed, %

There was also generated data for the percentage of correctly identified card ranks after corner voting. The data was generated by identifying 100 cards where all four corners were visible, and registering each time the card was correctly identified.

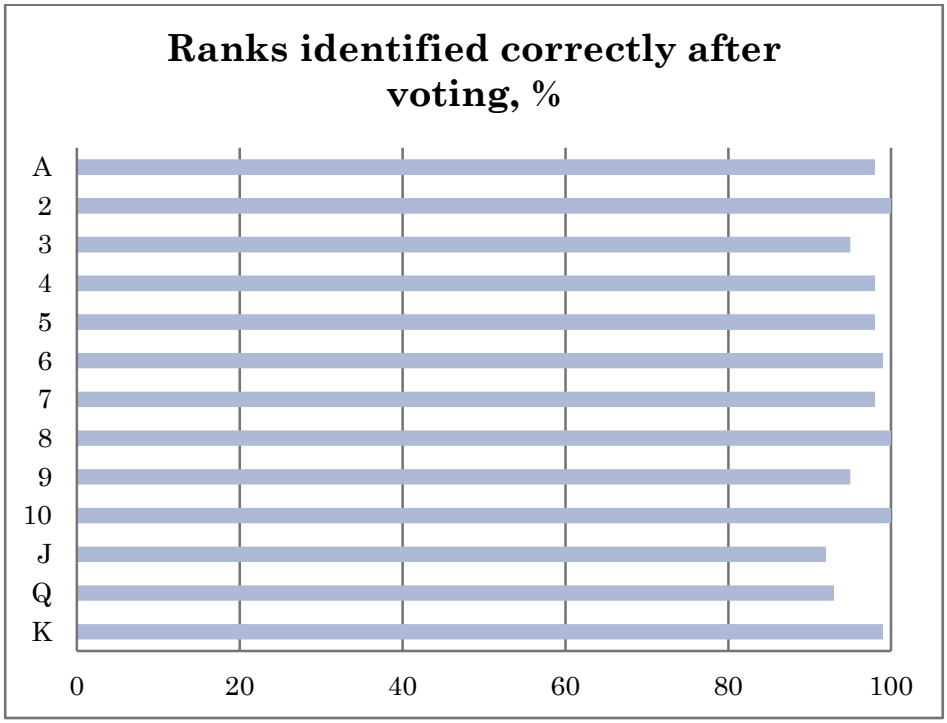


Chart 3: Ranks identified correctly after voting, %

3.7 Discussion

From the results, we can see that the `identifyLines` function was the main contributor to the big increase in processing time when more cards were added to the camera view. This was mainly due to calculating cosine and sine values when generating the Hough diagram. The problem was removed by instead pre calculating the values in the start of the program.

The improvement of processing time for the program was only continued until the time per frame got between one and two seconds, as this was the time considered necessary to use the program in the continued making of James the PokerBot. There still are several parts of the code that can and should be improved for further decreasing processing time per frame if the final product was to be commercialized.

As seen by Chart 2, the auto template generation not only made James independent on what card deck is used, but it also improved the accuracy of the correlation procedure

3.7.1 Further work

The computer vision program is still far from ideal. There are several improvements that should be taken care of, as well as some bugs that still have not been resolved.

One known bug is that the orientation of a card corner may be off by 180° if the card is placed at a 45° angle relative to the camera.

Other improvements that should be looked at are adapting parameters to the cameras position. Current parameters were to cover a wide variety of camera positions. This would lead to more falsely detected card corners than if the parameters were better adapted to the actual camera position.

The parameters mentioned above, include the detected size of the cards corner objects. This parameter should not only be defined by the camera position, but also the actual size of the object, as this can differ from card deck to card deck. Parameters like these, should be determined when auto generating templates for a card deck. It should then be possible to store information on different decks (templates and parameters).

When detecting the suit ID, this would currently only consider the shape of the object. In future versions, the color should be considered as well as this would separate between spade/club and heart/diamond.

The code for the computer vision program needs overall maintenance regarding design.

4 THE MAIN UNIT

The main module is what runs the computer vision and the Texas Hold'em applications used by James the PokerBot. The software for the main unit was initially developed on a stationary computer with an Intel® Core™2 Duo CPU running on 3GHz. Windows 7 was used as OS and Visual Studio 10 as IDE. For the prototype, this was cross compiled to run on an ARM based development board called the BeagleBoard. This board uses an OMAP3530DCBB72 720MHz processor and the Linux version "Ångström" was used as OS. To cross compile code, NetBeans was used as IDE.

4.1 Stationary

The final version of the main unit was never intended to run on a general-purpose computer. Still this was used for developing the applications that eventually would be running on the module. The reason for doing this was simply that there were a lot more experience programming under such circumstances.

Programs for the main unit were programmed in C++ and developed using Visual Studio while NetBeans was used to cross compile them to the BeagleBoard. The programs for the player modules were programmed in C, using AVR studio 5.1.

4.2 The BeagleBoard

The BeagleBoard measures approximately 75 by 75 mm and has all the functionality of a basic computer. The board uses up to 2 W of power and can be powered from the USB connector, or a separate 5 V power supply. Because of the low power consumption, no additional cooling or heat sinks are required.

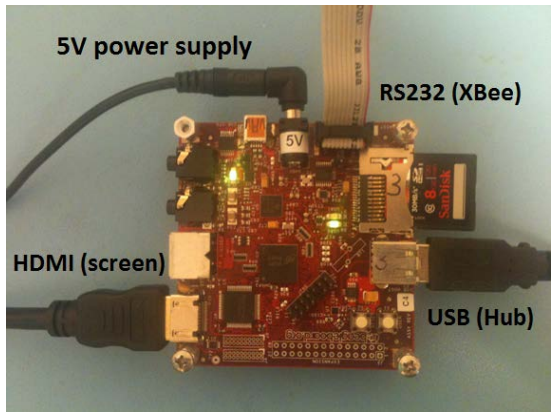


Figure 4-1: The BeagleBoard

For this project, the board would be connected to two cameras for the computer vision parts, an XBee for communication with the other players and a robotic arm for fetching the pocket cards belonging to James. It was also connected to a screen that was used for debugging. The screen also worked as a USB hub for connecting cameras, keyboard, mouse, USB-Ethernet adapter and power for one of the player modules.

4.2.1 Setting it up

By using [16], [17] and [18], we get the following procedure for setting up the BeagleBoard.

You will need:

- BeagleBoard
- SD card (minimum 4GB)
- A computer with internet connection and SD card reader
- Monitor (with DVI-D or HDMI)
- Power source (5.5x2.1mm Barrel Connector)
- USB to Ethernet adapter

Also you would want a keyboard and a mouse for when your BeagleBoard is ready for action!

4.2.1.1 Build angstrom image with opencv

To use Angstrom as OS for the BeagleBoard, it was necessary to build an image that would be stored onto the SD card. To build the image, the following procedure was used.

- Go to http://www.angstrom-distribution.org/narcissus/
- Use the following settings:
    - Base settings
        - Machine: beagleboard
        - Image name: your choice
        - Complexity: advanced
    - Advanced settings
        - Release (default): 2011.03
        - Base system (default): regular (task-base)
        - /dev manager (default): udev
        - Type of image: tar.gz
        - Software manifest: no
        - SDK type: simple toolchain
        - SDK hostsystem (default): 32bit Intel

- User environment selection: X11
- X11 desktop environment: leave all boxes blank
- Additional packages selections:
  - Development packages: OpenCV headers and libs
  - Additional console packages: All kernel modules, OpenCV
  - Network related packages: NetworkManager, NetworkManager GUI applet, Wireless-tools
  - Platform specific packages: OMAP Display Sub System (DSS)
    Documentation, BeagleBoard validation GUI extras, BeagleBoard validation GNOME image

- Click *"Build me!"* and wait
- Download the first "*.tar.gz" file. Don't worry about the second file

4.2.1.2 Set up the SD card

When the image is build, the next step is formatting the SD card correctly. For this, Ubuntu 11.10 was used for the development machine.

First find out where your SD-card is mounted. Plug in a SD-card into your computer wait for it to mount. Then run the following command.

```
df -h
```

You will see something like this:

```
/dev/sda5              98G   56G   38G  61% /
none                  1.5G  316K  1.5G   1% /dev
none                  1.5G  724K  1.5G   1% /dev/shm
none                  1.5G  336K  1.5G   1% /var/run
none                  1.5G     0  1.5G   0% /var/lock
none                  1.5G     0  1.5G   0% /lib/init/rw
none                   98G   56G   38G  61%
/var/lib/ureadahead/debugfs
/dev/sdb1              15G  8.0K   15G   1% /media/FAE3-DCE5
```

Look for a line that starts with /dev/sdXX in it; generally it will be mounted to the /media directory (on Ubuntu anyway). Make sure the size of the SD-card is right to verify. You will need this device directory when formatting the card.

For Angstrom to run, you will need to have an SD-card with two partitions on it. First one fat partition to hold the boot files and then an ext3 partition with the root file system on it. Thanks to Graeme Gregory there is a nice script to set up you SD-card. Execute the following commands which will make a working directory, download Graeme's script and execute it. Be sure to substitute your SD-card device directory (from above) for /dev/sdX below. Notice that if like in this example, the device directory for the SD cards is /dev/**sdb1**, you only add /dev/**sdb** in the line mentioned.

```
mkdir ~/angstrom-wrk
cd ~/angstrom-wrk
wget
http://cgit.openembedded.org/cgit.cgi/openembedded/plain/contrib/a
ngstrom/omap3-mkcard.sh
chmod +x omap3-mkcard.sh
sudo ./omap3-mkcard.sh /dev/sdX
sync
```

Remove the card, wait two/three seconds, plug it back in. Wait a few seconds and verify the SD-card is mounted by executing:

```
df –h
```

You should have two new mounted partitions:

```
/media/boot
/media/Angstrom
```

The card is now ready. You only need to do this process once for the life of the card.

4.2.1.3 Setup the boot partition

Now, first make sure the image that was downloaded above, is stored to your working directory. There are only three files that are mandatory for boot partition. The following lines will extract the files from the download build and copy those to the boot partition on the SD card.

```
# extract the files to the ./boot directory
tar --wildcards -xzvf [YOUR-DOWNLOAD-FILE].tar.gz ./boot/*

# copy the files to sc-card boot partition.
cp boot/MLO-* /media/boot/MLO
cp boot/uImage-* /media/boot/uImage
cp boot/u-boot-*.bin /media/boot/u-boot.bin
sync
```

Last thing to do is to copy the root file system.

```
sudo tar -xvz -C /media/Angstrom -f [YOUR-DOWNLOAD-FILE].tar.gz
sync
```

Now make it safe to remove the SD-card.

```
sync
umount /media/boot
umount /media/Angstrom
```

4.2.1.4 Boot the Beagleboard

The last thing to do now, is booting the BeagleBoard. Connect the BeagleBoard to a monitor using a HDMI cable, and then insert the newly formatted SD card to the card reader on the board. For this project, the monitor also doubled as a USB hub. This was done by connecting a USB cable from the BeagleBoard to the monitor. Then the keyboard, mouse, camera and USB to Ethernet adapter used later, could all be connected to the BeagleBoard via the monitor.

To boot, power up the board and then press the RESET button while holding down the USER button. Now you've got to be patient as the first boot takes a long time.

During this time, the LEDs USR0 and USR1 will be blinking. The screen will initially display a BeagleBoard logo. After a while it will go dark and say that the HDMI is unconnected. Then finally, it will display the Angstrom desktop.

To see if the internet connection is working, run the lightweight web browser Midori, found under Applications->Internet->Midori and see that you can connect to a web page like google.com.

4.2.2 Cross compiling

When the Beagle is up, running and online, programs can be cross compiled from another computer so that they will run on the BeagleBoard. For this task, Netbeans was chosen as IDE. Here code can be edited and debugged on your computer of choice while the code compiles and runs on the BeagleBoard itself.

To set up an environment where the programming and cross compiling would work seamlessly, a few steps had to be taken. On the Beagle, first update with “opkg update”, then the following packages had to be installed using "opkg install":

- bash
- task-native-sdk
- gdb

4.2.2.1 File mapping

For Netbeans to compile directly to the BeagleBoard, the file system on the board was mapped to the development machine (Windows 7) using samba.

The setup for doing this was found following a YouTube video [19]. A summary of this video is added in the following points.

- Create a password that will be used when connecting to the shared folder
  ```
  sudo smbpasswd -a root
  ```
- Open the samba configuration file
  ```
  sudo vim /etc/samba/smb.conf
  ```
- Make sure the workgroup settings matches the settings on the windows computer
- Go to the end of the document and insert the following configurations
  ```
  [sharedname]
  path = /realname
  available = yes
  valid users = techno
  read only = no
  browsable = yes
  public = yes
  writable = yes
  ```
  sharedname is the name seen by the windows computer. *path* contains the complete path to the folder you want to share.
- Next restart samba
  ```
  sudo /etc/init.d/samba restart
  ```
- Get the IP of the BeagleBoard using
  ```
  ifconfig
  ```

- On the Windows machine, open the run box by pressing the win button + R, and add
  ```
  \\IP-address\
  ```
- You should now be asked to input username and password. Unless you have done any changes, the username should be root and the password the one you chose in the procedure above.
- No you should be able to access the shared folder.

The procedure described above was used to share both the entire file system of the BeagleBoard and to the folder that would contain the NetBeans projects. These were then mapped as local network drives on the windows computer. This was done so that they would be easily accessed by NetBeans when cross compiling. The file system (used to access include directories on the beagle) was mapped to X and the program folder to Y.

Now for setting up NetBeans, the procedure described in [20] was used.

- First, add the BeagleBoard as a new remote development host. This was done by going to “tools->options” then clicking the "C/C++" tab and then the "Build Tools" sub-tab. Now click the "Edit" button on the right.
- Click "Add" to add a new development host. Enter the IP address of the BeagleBoard and click “Next”.
- Again enter root your password as login details for the Beagle and click “Next”. NetBeans will then connect by SSH to the Beagle and run a script to find the development tools (GCC compiler, linker, make etc.).
- Again click "Next", then ensure that "File system sharing" is selected under "Synchronization" and click "Finish".
- Now you should get see that the BeagleBoard is added as a host

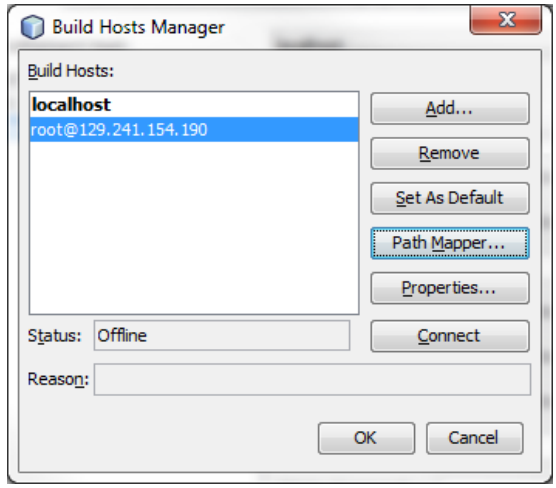


Figure 4-2: Screenshot NetBeans

- Click "Path Mapper" and add the information for the shared folder from the Beagle

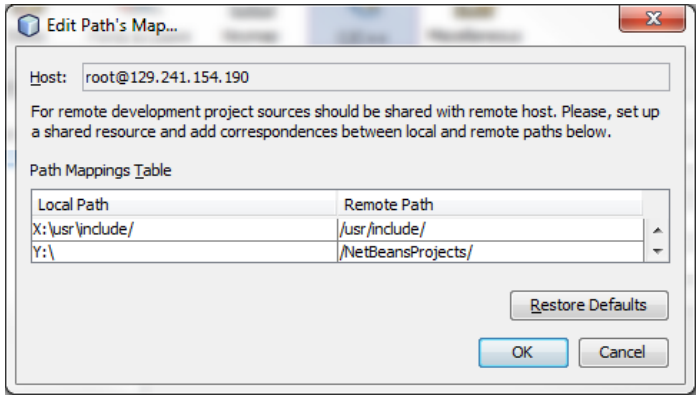


Figure 4-3: Screenshot NetBeans

- Next is adding necessary include directories. First the default directories were removed. Then for the Texas Hold'em and computer vision applications in this project, the following was included.

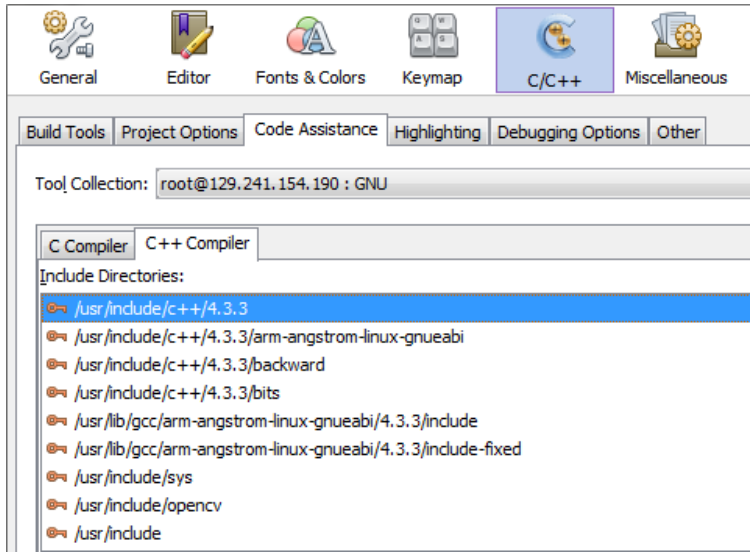


Figure 4-4: Screenshot NetBeans

- Finally all is ready to start making a new project. For this project, a new C/C++ Application was made and the BeagleBoard was set as development host.

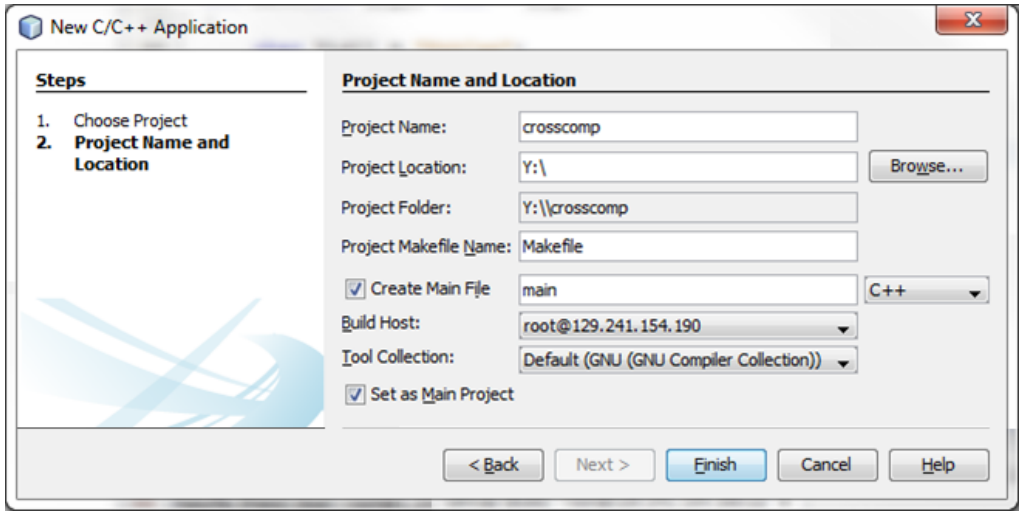


Figure 4-5: Screenshot NetBeans

- NetBeans will auto generate a Makefile, but to get openCV to work, the following flags had to be set aswell

```
CPPFLAGS = -g -Wall -Wno-unused-function `pkg-config --cflags opencv`
LDFLAGS = `pkg-config --libs opencv`
```

- To test that all the settings worked, a camera was connected to the BeagleBoard and the following program was added to the main.cpp file of the new project.

```
#include <cstdlib>
#include <iostream>
#include <opencv/cv.h>
#include <opencv/highgui.h>

using namespace std;

int main(int argc, char** argv)
{
      cout << "Hello World" << endl;

      char Vid[] = "WebCam";
      IplImage * frm;
      CvCapture * capture;

      capture = cvCaptureFromCAM(CV_CAP_ANY);

      if( !capture ) {
            cout << "Cannot Open the Webcam" << endl;
            return 0;
      }

      cvNamedWindow (Vid,1);

      do {
            frm = cvQueryFrame( capture );
            if( frm )
                  cvShowImage (Vid, frm);

            int k = cvWaitKey( 50 );
            if (k == 27 || k == '\r') // Press ESC or Enter
                  break;                    // for out

      } while( frm );

      cvDestroyWindow( Vid );
      cvReleaseCapture( &capture );
      return 0;
}
```

This project should fetch frames from the camera connected to the Beagle and display these in a window called “WebCam”

- In NetBeans, press F11 to compile. Then on the beagle, open a terminal and type

```
cd pathToFile
./program
```

  For this example, pathFoFile = “/NetBeansProjects/crosscomp/dist/Debug/GNU-Linux-x86” and program = “crosscomp”.
- See that the program is working, and you are ready to make your application.

4.2.3 The computer vision and Texas Hold’em applications

When cross compiling programs for the main module, the computer vision program did not need any alteration to compile except for changing the location of the templates used. However there was a problem regarding speed when running this program on the BeagleBoard. The problem was fixed, and how this was done is described in chapter 3.2.

For the Texas Hold’em application on the other hand, it was necessary to do quite a bit of alternating for it to compile to the BeagleBoard. This was mostly due to the way it communicated with the player modules. On the stationary computer, this was done using virtual com ports that was accessed using managed C++ (System::SerialPort). On the Linux based BeagleBoard, communication was achieved using Zigbee via XBee modules that were connected to the serial port on the board. This port was then accessed through /dev/ttyS2. This is further described in chapter 6, Communication.

4.3 Camera

The camera used for input to the computer vision program, was a Microsoft LifeCam HD-6000. The main reasons for choosing this camera were that it was cheap and available. When using it with the computer running Windows, it worked without any problems. When using it with the BeagleBoard using Angstrom, camera initialization would generate the following messages.

```
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
```

This did not affect the functionality of the program, but it was observed that the frames grabbed would now come from a buffer of seven frames. When the processing time of each frame was roughly 1-2 seconds, the response of the program would be delayed by 7-14 seconds. When retrieving a new frame that would be processed, this delay was minimized by first discarding five frames. Discarding the five outdated frames would result in a new delay of almost a second as the frame grabbing itself very slow.

Figure 4-6: Cameras used, Microsoft LifeCam HD-6000 and PS3 eye

To see if the problems were due to the camera or something else, a bit of research was done on the subject. A camera that then was found to be recommended on several online forums for use with the BeagleBoard was the PS3 eye. The computer vision program was therefore tested using this camera, to see if this would remove the “warnings” and increase the frame grabbing speed.

The cameras were tested using the same speed measuring technique as described in chapter 3.2. As the delay problem was due to grabbing the five outdated frames, this was also the amount of frames used for the comparing. The following code was then used to generate the length of the delay.

```
// time for grabbing 5 frames
t1 = clock();
frm = cvQueryFrame( capture );
frm = cvQueryFrame( capture );
frm = cvQueryFrame( capture );
frm = cvQueryFrame( capture );
frm = cvQueryFrame( capture );
t2 = clock();
diff = (float)(t2 - t1) / CLOCKS_PER_SEC;
cout << "Total time: " << diff << endl;
```

When using the Microsoft LifeCam, the following output was returned.

```
root@beagleboard:/NetBeansProjects/texasholdem/dist/Debug/GNU-
Linux-x86# ./t
exasholdem
Hello World!
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
VIDIOC_QUERYMENU: Invalid argument
Total time: 0.99
Total time: 0.89
Total time: 0.88
Total time: 0.89
Total time: 0.89
Total time: 0.89
Total time: 0.9
Total time: 0.89
```

We here see the mentioned “warnings” (`VIDIOC_QUERYMENU: Invalid argument`) as well as the delay being found at 0.9 seconds.

For the PS3 eye, the generated output was:

```
root@beagleboard:/NetBeansProjects/texasholdem/dist/Debug/GNU-
Linux-x86# ./t
exasholdem
Hello World!
Total time: 0.28
Total time: 0.18
Total time: 0.18
Total time: 0.19
Total time: 0.18
Total time: 0.18
Total time: 0.18
Total time: 0.18
Total time: 0.18
```

We can see how all of the warnings have disappeared and how the delay has decreased till 0.2 seconds.

Still, the PS3 eye could not be used for this project as it did not have automatic brightness setting, as well as blurring the image too much. The configurations of the camera then lead to a completely unacceptable input for the computer vision program. This can be seen in the following pictures.

Figure 4-7: Frame from the Microsoft LifeCam

Figure 4-8: Frame from the PS3 eye

It was concluded that the delay generated by the Microsoft LifeCam was acceptable for further developing the PokerBot, and now additional cameras were tested.

The plan was that James the PokerBot would be using two cameras. The first one would be the main camera, used to identify and locate playing cards on the table. The second one would be located beneath the table edge. This would then be used to read the private pocket cards retrieved by the robotic arm. As the robotic arm was never completed, neither was the installation of the second camera.

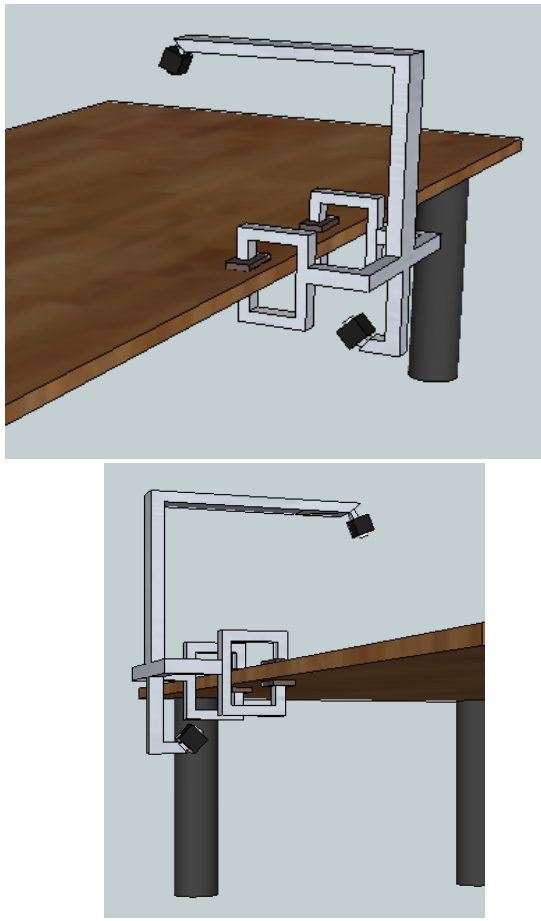


Figure 4-9: Camera positions

4.4 Communication device

For James to be able to play against human players, some form of communication between the main unit and the player modules was necessary. On the stationary computer, virtual COM ports were used to achieve this. Simulated player modules would then use these ports to communicate with the Texas Hold'em application. When running the Texas Hold'em application on the BeagleBoard, ZigBee devices called XBee were used. These were connected to the BeagleBoard through the RS232 header on the board. More about this can be found in chapter 6, Communication.

4.5 Robotic arm

The plan was that James would be connected to a robotic arm that would be used to fetch playing cards. This arm was never completed but ideas on how it could have been made are discussed in chapter 8.

5 THE PLAYER MODULES

When playing Texas Hold'em, a full table would normally consist of up to 9 or 10 players. These players would have to interact with James through custom made player modules. The plan was that these modules also would be designed as downloadable applications for use with smartphones and tablets.

For this project, three player modules were made. The first was a simulator, made as a windows form application while the other two were physical prototypes based on AVR processors. A module would generally consist of a small display, 6 buttons and a communication device. The display is used to output game status. This includes the current pot, the players bankroll, the high bet of the round and the players current bet. The buttons would have the following functionality

- Fold
- Check/Call
- Bet/Raise
- Up (increases bet)
- Down (decreases bet)
- Menu

The communication devices used in this project were virtual COM ports for the simulator programs and ZigBee modules for the prototypes. These would provide a way for the player module to communicate with the main module and the way this was achieved, is further described in chapter 6.

5.1 Simulator

When developing the Texas Hold'em application, it was necessary to get input from all the players in the game. The application was supposed to run on a BeagleBoard, but was initially developed on a stationary computer. It was then created a small program that would simulate player modules, to provide the needed input.

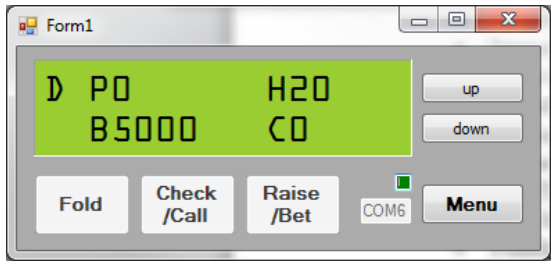


Figure 5-1: Player module simulator

The simulator was programmed in C++ using a windows form application. The form was designed so it would look similar to the way expected by the physical modules. As communication device, a small text box was used to specify the name of a virtual COM port. The port needed to be unique for each instance of the simulator program running and would then be connected to the Texas Hold'em application. There was also added a small box to simulate a led for indicating whether the module was active or not.

5.1.1 The tools

The windows form consisted of the following tools:

- 6 Windows::Forms::Button
- 1 Windows::Forms::Label
- 2 Windows::Forms::TextBox
- 1 IO::Ports::SerialPort

5.1.1.1 Variables

The following variables were used to store the player's game stats.

```
int state;        // Used to print N-neutral, S-smallBlind, B-
bigBlind, D-Dealer
int highBet;      // The highest bet on the tabel/the bet to match
int curBet;       // The bet you have currently added to the pot
this round
int bet;          // The amount you want to add to your current bet
this bid round
int bigBlind;     // Used when increasing the bet
int pot;          // Current amount of chips in the pot
int bankRoll;     // Amount of chips in the players bank
```

5.1.1.2 The buttons

The six buttons could be divided into three groups.

- The action buttons
  - These consist of the “Fold”, “Check/Call” and “Raise/Bet” buttons.
    The function of these buttons, is informing the main unit about what actions have been taken by the player. The buttons would be enabled when a player is allowed to perform an action that is when the player is the talker (see appendix 13.1 for poker rules). When clicking one of them, all the action buttons will be disabled as the player no longer will be the talker.

- The bet buttons
  - This consists of the “up” and “down” buttons.
    These worked by raising/lowering the wanted bet by the value of the bigBlind, without going above the bankroll or below 0.

- The menu button
  - For the simulator, this button was never implemented. The plan however, was that it would be used to start/stop/pause the game, add new players, look at the game stats of a different player, see the cards detected on the table, tell James if he identified the cards wrongly etc.

5.1.1.3 The label

The label works as the “display” on the player module. It has room for 2x16 characters as this was the resolution of the display planned to use for the physical prototype. The display would print the game stats for the player the following way:

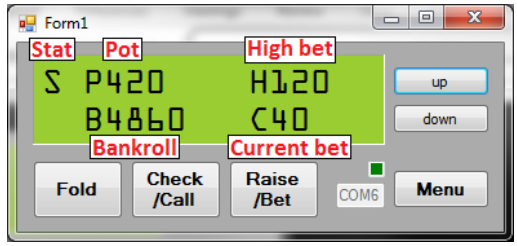


Figure 5-2: Player module, display output explained

Line 1, char 1: The player's state. D: dealer, S: small blind, B: big blind, N: neutral.

Line 1, char 3-9: The current value of the pot.

Line 1, char 10-16: The highest bet given this round. This is the bet the player must match or excide to be allowed to continue the round.

Line 2, char 3-9: The value of the players bank roll.

Line 2, char 10-16: The current bet the player has added this betting round.

The display was also supposed to show the information described to be accessible from the menu button as explained in the previous subchapter.

5.1.1.4 The text boxes

The form consists of two textboxes. The larger one contains the name of a COM port that is used to interact with the Texas Hold'em application. When starting the simulator, this textbox will be enabled and the name of the required COM port can be entered. When leaving the box afterwards, the text is added as the name of the serial port, the serial port is opened and the text box is disabled.

The second textbox was only meat to simulate a led, indicating whether the module is active or not. It was made into a text box, only to have a way to "leave" the larger box with the COM port name, without having to click on one of the forms buttons.

5.1.1.5 The serial port

As mentioned, the serial port was used to communicate with the Texas Hold'em application. The main function for the simulator was the event handler for receiving data on this port. When the serial port has opened, the simulator will wait for an event to be raised when data is received on the port. This event will react according to the command received. Examples of this can be updating the game stats, activating the action buttons or displaying additional information on the screen.

The code for actions taken when receiving data on the port is added in the appendix 13.2.2.

5.2 Hardware prototype

When the computer vision and Texas Hold'em applications were compiled to run on a BeagleBoard (chapter 4.2), two prototype player modules were made. Both of these were based on AVR processors and would communicate with the main module using ZigBee.

5.2.1 E-Blocks

The first prototype was made using the following E-Blocks from Matrix multimedia.

- Atmel AVR® multiprogrammer system, EB194-00-2
- E-blocks LED board, EB004
- E-Blocks LCD board, EB005
- E-Blocks push-to-make switch board, EB007

As well as these E-Blocks, an XBee module from Digi International was connected to the ATmega's UART for communication with the main module.

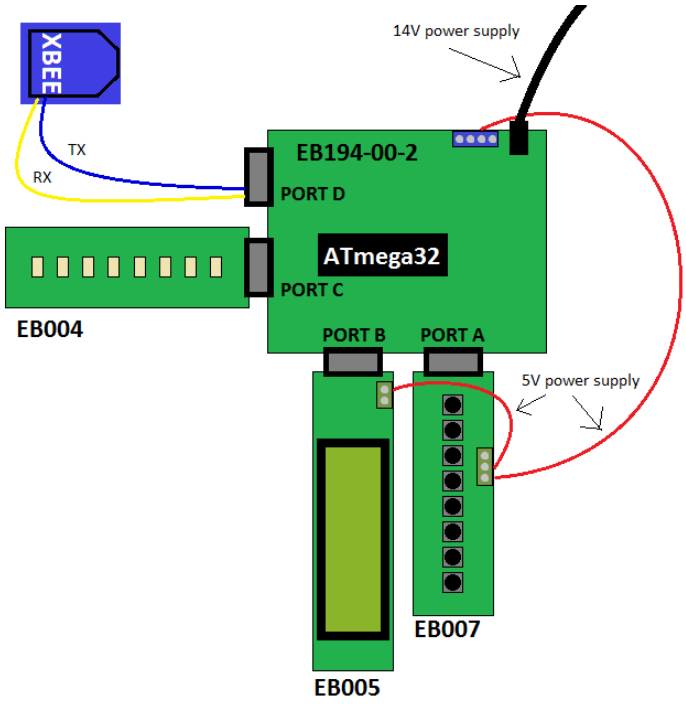


Figure 5-3: Rough overview of prototype 1

5.2.1.1 Programming

The ATmega32 was programmed in C, using AVRStudio 5.1. This is an IDE developed by Atmel, which is the same company as the one making the ATmega. The main function for the module was implemented as follows:

```
DDRC = 0xff; // Set port C as output ( LED E-Block )
DDRA = 0x00; // Set port A as input ( switch E-Block )

LCD_init( );
xBee_Init( );
player_initPlayer( );

LCD_putstr( LINE1, "Getting player" );
while ( !registerAsPlayer( ) );

LCD_putstr( LINE1, "Ready to play" );
while(1) {
      while( !buttonDown( ) && !xBee_DataInReceiveBuffer( ) );
      PORTC = 0x00;
      if (xBee_DataInReceiveBuffer( ))
            handleInput( );
      else{
            _delay_ms( 10 ); // de bounce
            handleSw( );
      }
      while( buttonDown( ) ); // Wait for button to go up again
      if( thePlayer.active == 1 )
            PORTC = 0xFF;
}
return 0;
```

5.2.1.2 Main Block – EB194-00-2, Atmel AVR® multiprog system

This board would connect to a PC via ISP for programming the Atmel AVR microcontroller, ATmega32, located on the board. This microcontroller was the programmed to control the following E-Blocks to work as a player module.

1. Power connector 2.1mm
2. Bridge rectifier to accept any polarity from power connector
3. Output voltage screw terminals
4. Power indicator LED
5. 5 volt voltage regulator
6. 28-pin AVRISP programming header
7. 8-, 20-pin AVRISP programming header
8. 40-pin AVRISP programming header
9. Microcontroller reset switch
10. Removable crystal to drive the microcontroller clock.

11-14. 28-, 20-, 8-, 40-pin AVR microcontroller socket

15. Expansion header containing the complete AVR microcontroller I/O

16-19. Port D-A I/O connector

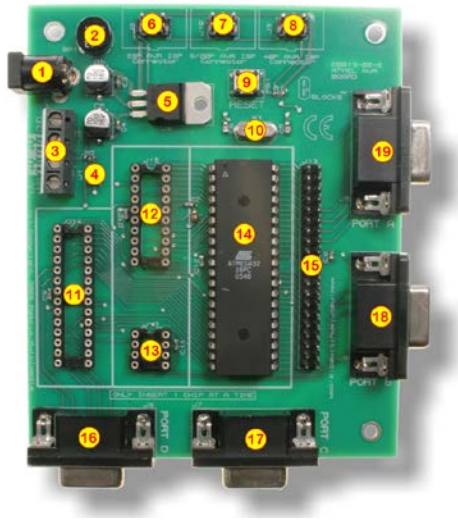


Figure 5-4: Board layout

5.2.1.3 LCD display – EB005

The LCD display used was connected to port B on the ATmega. It would display output in the same manner as described for the simulator application (chapter 5.1.1.3). The EB005 board used had the following layout:

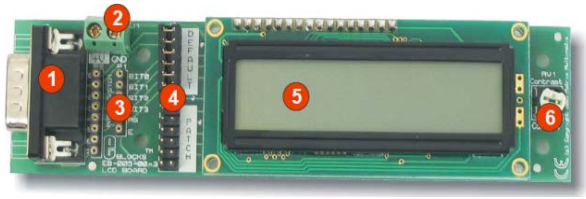


Figure 5-5: EB005, Board layout

1. 9-way downstream D-type connector
2. Power screw terminals
3. Patch connectors
4. Connection selection blocks
5. LCD display
6. Contrast potentiometer

The board was powered connecting a 5V power supply from the main board (EB194-00-2) to the power screw terminals on the EB005. Ground was connected through the 9'Th pin on the D-Sub.

The main function made to interface with the board, was called LCD_putstr. This function would take two parameters. The first parameter would specify what line the text, given as parameter two, would be printed on. The function is added in appendix 13.2.3.1. The rest of the code used by the LCD display is added in the digital attachment under *BeagleBoard – Player modules – E-Blocks – header*: *lcd.h* and – *src*: *lcd.c*.

Figure 5-6: Example game stat output for player

5.2.1.4 Switches and LEDs – EB007 and EB004

The switch board was used to get input from the players. The switches worked like generally described for a player module.

The main function for handling input from the switches was called handleSw and is added in appendix 13.2.3.2. Here two functions called handleUpSw and handleDownSw are used. These will increase/decrease the bet of the players with increasing speed as long as the button is held down and will set the bet to max/min of what the player can bet if double clicked. This and other switch functions are added in the digital attachment under *BeagleBoard – Player modules – E-Blocks – header*: *switches.h* and *src*: *switches.c*.

Figure 5-7: EB007, Board layout

1. 9 Way D-type Plug
2. 8 x Switches SW0 – SW7
3. Screw terminal
4. 9 Way D-type socket

Like for the EB005, the EB007 was also connected to a 5V power supply from the main board (EB194-00-2) to the screw terminals as well has connecting to ground over the 9'Th pin on the D-Sub.

The LED board was used for debugging, displaying whether or not the module is active in the game as well as if unread data is received on the UART. The LEDs were directly accessed through PORTC on the ATmega.

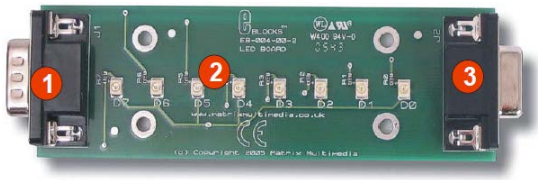


Figure 5-8: EB004, Board layout

1. 9 Way D-type Plug
2. 8 x LEDs D0 – D7
3. 9 Way D-type socket

5.2.1.5 Complete E-Block setup for prototype1

The E-Blocks were connected to each other as displayed in Figure 5-9. The main block, EB194-00-2, was powered using an external 14V power supply.

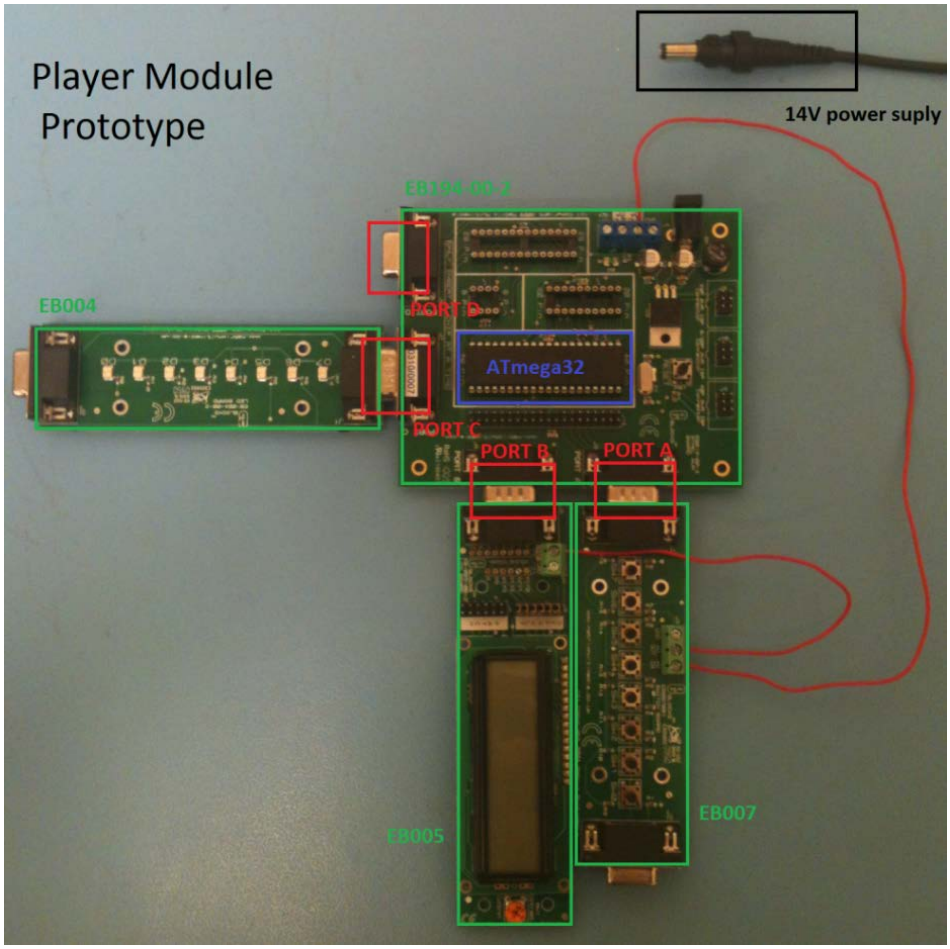


Figure 5-9: Complete E-block setup

Port D on EB194-00-2 would be used to connect to the XBee module used for communication. This is further described in chapter 6.

5.2.2 XMEGA-A3BU Xplained

To test that the Texas Hold'em application worked with more than one human player, a second prototype was made. This was based on the XMEGA-A3BU Xplained kit from Atmel. The kit includes, among others, the following peripherals.

- FSTN LCD display with 128x32 pixels resolution
- Three mechanical buttons
- One Atmel AVR QTouch® button
- Two user LEDs
- Four expansion headers

An XBee from Digi International was connected to the ATxmega's UART using the expansion header J1 on the board. The XBee was used for communication with the main module and this is further described in chapter 6.

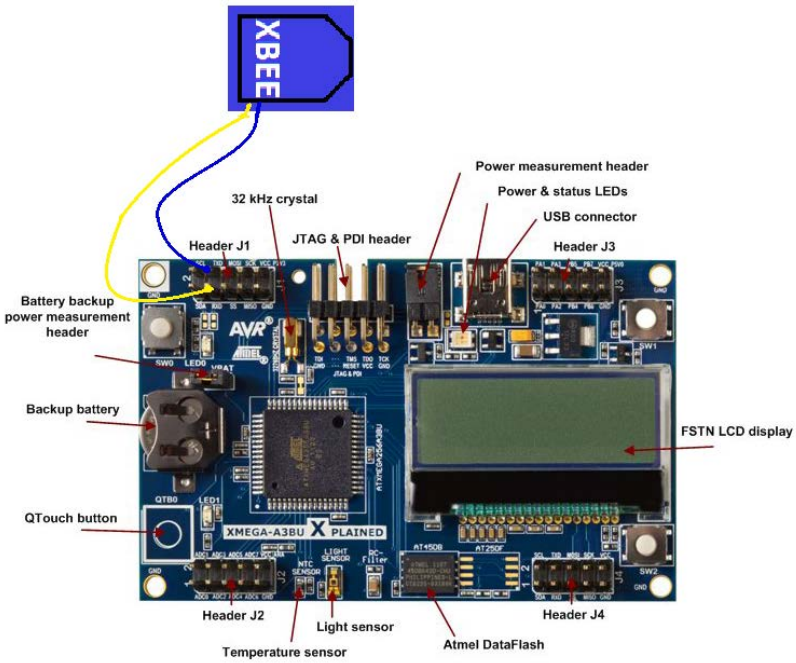


Figure 5-10: A3BU + XBee module

5.2.2.1 Programming

Like the E-Blocks, the A3BU was also programmed in C using AVRStudio 5.1. When making the code for the second prototype, most of the code already made for the first prototype could be reused. Some exceptions were regarding the LCD display, button/LED interface and enabling interrupts.

The main function for the module was implemented as follows:

```
board_init( );
sysclk_init( );
LCD_init( );
ioport_set_pin_high( NHD_C12832A1Z_BACKLIGHT );
PMIC.CTRL = PMIC_HILVLEN_bm;
xBee_Init( );
player_initPlayer( );

LCD_putstr( LINE1, "Getting player" );
while ( !registerAsPlayer( ) );

LCD_putstr( LINE1, "Ready to play" );
while(1) {
      while( !buttonDown( ) && !xBee_DataInReceiveBuffer() );

      if ( xBee_DataInReceiveBuffer( ) )
            handleInput( );
      else{
            _delay_ms( 10 ); // debounce
            handleSw( );
      }
      while( buttonDown( ) ); // Wait for button to go up again

      if(thePlayer.active == 1)
            ioport_set_pin_high( NHD_C12832A1Z_BACKLIGHT );
      else
            ioport_set_pin_low( NHD_C12832A1Z_BACKLIGHT );
}
```

5.2.2.2 LCD display

As mentioned, the biggest changes done to the code from the first prototype, was regarding the LCD display. In the old code, the display was accessed using a self-made library. As the display on the A3BU was a different type, this code could not be reused as it was. Therefore, the function used to interface with the display was updated to using a premade library from Atmel. This was achieved using a template made for the A3BU when generating the project. The code needed by the LCD was then found in the file *gfx_mono_text.c*, were the following functions were used:

- gfx_mono_init – used for initializing the display
- gfx_mono_draw_char – used to output a single char

All code used by the display is added to the digital attachment under *BeagleBoard – Player modules – A3BU – header*: *lcd.h* and – *src*: *lcd.c*

As the display on the A3BU has resolution of 128x32 pixels and the font used had a character size of 6x8 pixels, only half of the display was needed to print the same information as in the first prototype. It was therefore created a custom font to display detected community cards on the lower half of the screen. This font included the playing card symbols for club, heart, diamond and spade printed I two different sizes, 15x15 and 11x11 pixels. Code needed to make this font is added to the digital attachment under *BeagleBoard – Player modules – A3BU – header*: *cardFont.h*, – *src*: *cardFont.c* and – *src – config*: *conf_ cardFont.h*.

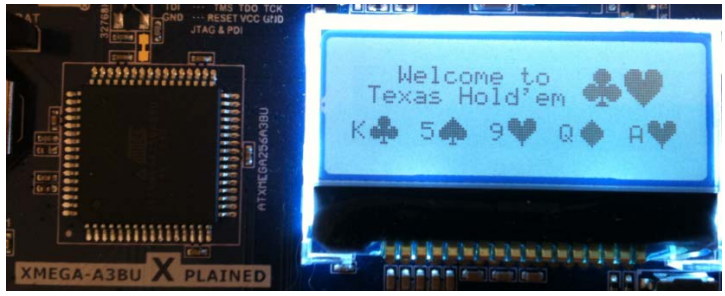


Figure 5-11: Displaying custom made fonts

Figure 5-11 displays an example output. The text "Welcome to Texas Hold'em takes up about the same space as was needed to display player stats the way it was done for prototype 1. The heart and club from the lager card font is displayed in the upper half of the display. All symbols from the smaller card font are displayed in the example game board printed on the lower half of the display. In this case, the flop consists of king of clubs, five of spades and nine of hearts, the turn is the queen of diamonds and finally the river here given as the ace of hearts.

5.2.2.3 Buttons

While a standard player module would contain 6 buttons, the A3BU only had 4. The "up" and "down" buttons (used to increase/decrease the bet) were therefore not included for this prototype. External buttons could have been added using the expansion headers on the board, but doing this was not prioritized as these buttons had already proved to work for the first prototype.

Another change that was done to the old code regarding button handling, was replacing `switch(PINA)` in the button handling with `switch(getButton())`.

The getButton() function and the defines used by the function, are added in appendix 13.2.3.3.

6 COMMUNICATION

As there normally are more than two players in a round of Texas Hold'em, it was necessary to use some form of point-to-multipoint communication. For the simulator program, this was achieved by setting up virtual COM port pairs to communicate with the Texas Hold'em application. The prototyped player modules would use ZigBee to communicate with the main module.

6.1 Virtual COM ports

The virtual COM ports were made using a program called com0com. This is a Null-modem emulator that would create the ports on the computer running the player module simulators and Texas Hold'em application.

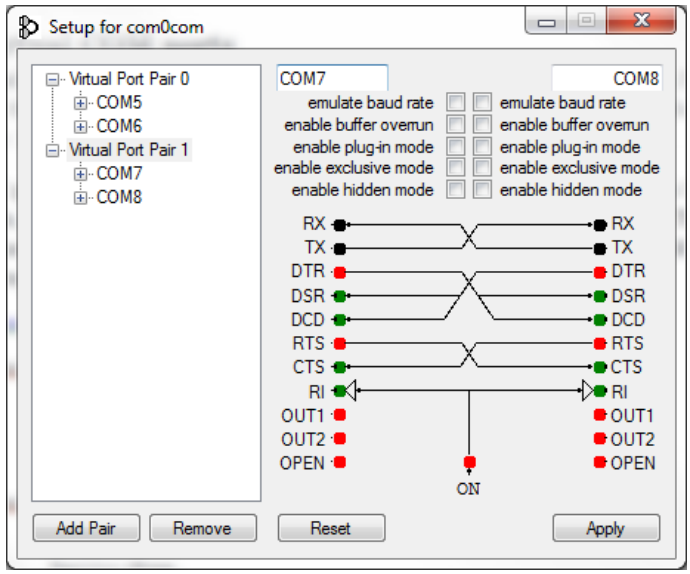


Figure 6-1: com0com setup window

The COM ports were created manually before starting the game. In the Texas Hold'em application, premade COM ports were hard coded as an array of serial ports. The number of players that could connect to the game was then limited to the amount of COM ports added to this array.

```
array<SerialPort ^> ^ serialPorts = gcnew array<SerialPort ^>(2);

serialPorts[0] = gcnew SerialPort(
      L"COM5",
      9600,
      Parity::None,
      8,
      StopBits::One);
serialPorts[1] = gcnew SerialPort(
      L"COM7",
      9600,
      Parity::None,
      8,
      StopBits::One);
```

Both creating the virtual COM ports and making the array of available ports could be achieved in runtime by the Texas Hold'em application, and would therefore open up to the possibility of playing against an undefined number of human players. As the COM port solution only was used for developing the Texas Hold'em application, it was not prioritized to improve this.

6.2 ZigBee

It was desired that the finished version of James the PokerBot, would communicate with player modules using a wireless standard. For the prototype, ZigBee was chosen as it is intended to be simpler and less expensive than other WPANs, such as Bluetooth.

ZigBee is a specification for a suite of high level communication protocols using small, low-power digital radios based on an IEEE 802 standard for personal area networks [21].

6.2.1 XBee®

XBee modules were chosen to set up the wireless network as they are very cheap, easy to use and designed to work with point-to-multipoint communication. There exist several different versions of the XBee module. For this project, "XBee 1mW PCB Antenna – Series 1" modules form Digi International were used.

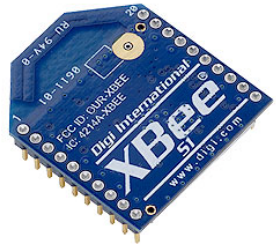


Figure 6-2: XBee 1mW PCB Antenna – Series 1

6.2.1.1 Theory

The following subchapter will give a short introduction to the basics of an XBee module. The data is retrieved from [22].

The XBee®/XBee-PRO® RF Modules interface to a host device through a logic-level asynchronous serial port. Through its serial port, the module can communicate with any logic and voltage compatible UART; or through a level translator to any serial device (For example: Through a Digi proprietary RS-232 or USB interface board).

Specifications

Range:

- Indoor/Urban: up to 100' (30 m)
- Outdoor line-of-sight: up to 300' (90 m)
- Transmit Power: 1 mW (0 dBm)
- Receiver Sensitivity: -92 dBm

Power:

- TX Peak Current: 45 mA (@3.3 V)
- RX Current: 50 mA (@3.3 V)
- Power-down Current: < 10 µA

UART Data Flow

Devices that have a UART interface can connect directly to the pins of the RF module as shown in the figure below.

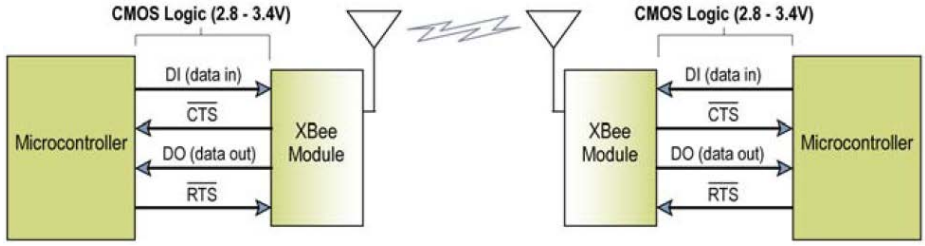


Figure 6-3: System Data Flow Diagram in a UART-interfaced environment

Serial Data

Data enters the module UART through the DI pin (pin 3) as an asynchronous serial signal. The signal should idle high when no data is being transmitted.

Each data byte consists of a start bit (low), 8 data bits (least significant bit first) and a stop bit (high). The following figure illustrates the serial bit pattern of data passing through the module. Example Data Format is 8-N-1 (bits - parity - # of stop bits)

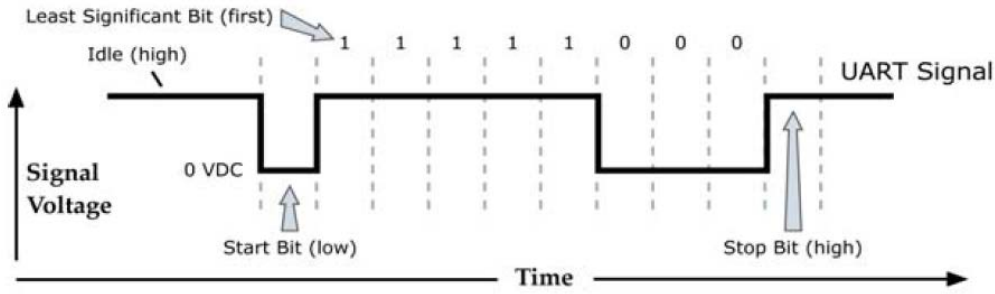


Figure 6-4: UART data packet 0x1F (decimal number "31") as transmitted through the RF module

Serial communications depend on the two UARTs (the microcontroller's and the RF module's) to be configured with compatible settings (baud rate, parity, start bits, stop bits, data bits). The UART baud rate and parity settings on the XBee module can be configured with the BD and NB commands, respectively. More about this can be found in chapter 6.2.

6.2.1.2 Series 1 vs. Series 2

The XBee Series 1 and the XBee Series 2 modules have the exact same form factor and are pin-for-pin compatible, but they are based on different chip sets and are running different protocols, so they are not over-the-air compatible. The Series 1 module is based on the Freescale chipset and is intended to be used in point-to-point and point-to-multipoint applications. The Series 2 module is based on the Ember chipset and is designed to be used in applications that require repeaters or mesh. Both modules have the option to interface via AT or API modes and both series will be offered and fully supported moving forward [23].

In this project, all the modules using the XBees would be located around a table. It was therefore not considered to be requiring repeaters or mesh, but that it would do fine with point-to-point communication, which is compatible with the series 1 module.

6.2.1.3 Normal vs. PRO

There are a few differences between the regular XBees and the XBee Pros. The Pros are a bit longer, they have a longer range (indoor: 90m vs. 30m, outdoor: 1600m vs. 90m) use more power (transmit current: 250mA vs. 45mA, idle: 55mA vs. 50mA) and cost more money ($37.95 vs. $22.95 at sparkfun.com, 12-05-2012). Once again, as all the modules would be oriented around a table, a 30 meter indoor range was considered enough, and the normal version vas used. (Data on range and power found here, [22])

6.2.2 Setting up XBees to work with the main – and player modules

During developing, the XBees were installed on a breadboard. To do this, the 2mm pin spacing on the XBees had to be converted to 0.100” spacing using an XBee adapter board.

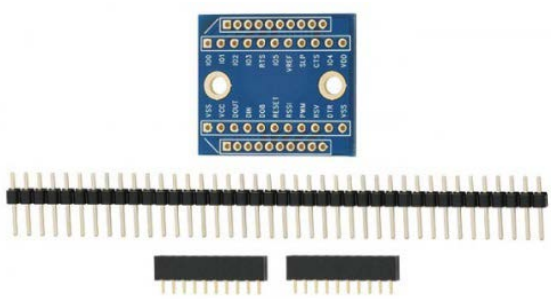


Figure 6-5: XBee Adapter Board

From the XBee manual [22], we see that it is recommended to reduce noise from the supply voltage by placing a 1.0 µF and 8.2 pF capacitor as near as possible to the VCC pin on the XBee. As these capacitors were not available when testing the system so a .1 µF capacitor was used instead.

The XBees operate at 3.3V, so the input and output signals had to be converted to match the module using the XBee. A close description as well as schematics on how this was done for each module, is given in the following part chapters. Figure 6-6 provides an overview of the total setup on the breadboard.

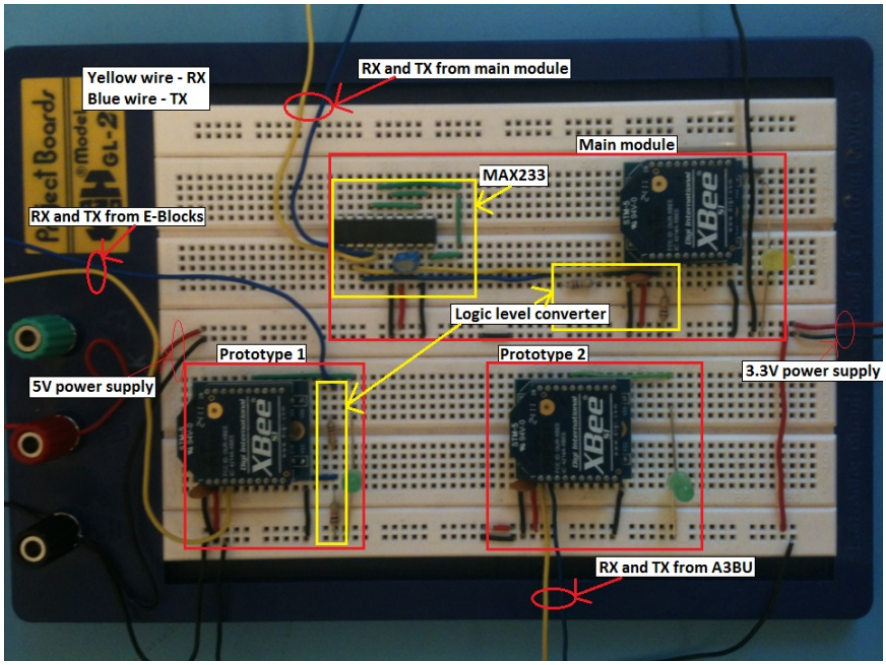


Figure 6-6: XBee interface to main and player modules

The LEDs are connected to pin 15 on the modules. This will provide a status LED that will flash differently depending on the state of the module

6.2.2.1 Main module

For the main module, the XBee was connected to the RS232 header on the BeagleBoard. It would transmit a signal of ±5V and expect to receive one of ±5V-±25V. The XBee was therefore connected to the main module via an MAX233 circuit to convert the signals. As this driver would transmit a 5V signal to the XBee, a simple logic level converter was made to convert this signal down to 3.3V so that the XBee would not be damaged.

It would be better to use a MAX232SOIC16 instead of the MAX233, as this operates at 3.3V, which the same voltage as the XBee. It would then not be necessary to convert between 5V and 3.3V.

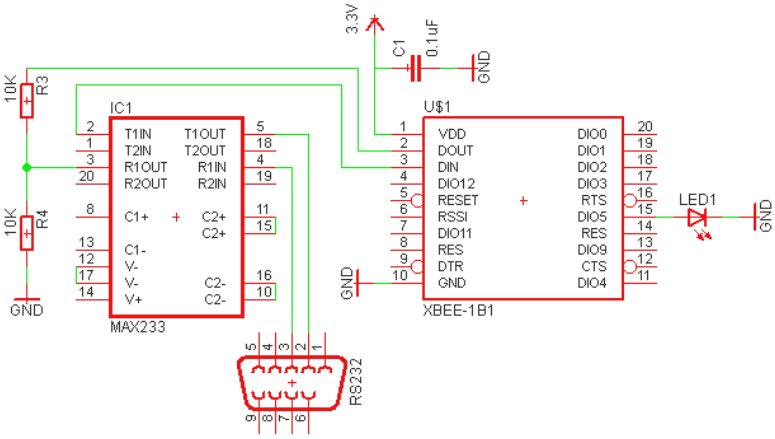


Figure 6-7: Schematics for connecting an XBee to the main module

Port setup

The following code is the setup used to initialize the serial port.

```
class xBee {
private:
    struct termios tio;
    int ttyFd;
(…)
}

void xBee::setupPort( )
{
      bzero( &tio, sizeof (tio) );
      tio.c_iflag = IGNPAR;
      tio.c_oflag = 0;
      tio.c_cflag = B9600 | CRTSCTS | CS8 | CLOCAL | CREAD; // 8n1,
baud rate 9600
      tio.c_lflag = 0;

      /* read() will always return immediately; if no data is
available
      it will return with no characters read. */
      tio.c_cc[VMIN] = 0;
      tio.c_cc[VTIME] = 0;

      ttyFd = open( "/dev/ttyS2", O_RDWR | O_NOCTTY );
      tcflush( ttyFd, TCIFLUSH );
      tcsetattr( ttyFd, TCSANOW, &tio );
}
```

Data could then be read and written using the following functions from *unistd.h*;

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

The functions will read/write up to *count* bytes from file descriptor *fd* into/from the buffer starting at *buf*.

Problem using /dev/ttyS2 with Angstrom

When the port was configured to work with the XBee modules, there was still a problem getting communication to work properly.

The serial port connected to the RS232 header on the BeagleBoard, was accessed through `/dev/ttyS2`. The problem would appear when attempting to read data from the port. It was then observed that the data was retransmitted directly to the TX pint on the RS232 header, without actually being read. If data was sent to the port in a loop by crossing the RX and TX pins, it would eventually be read. It was assumed that this came from some form of automatic synchronizing. The problem was that if no data was sent a while after the port had synchronized, it would need to be re synchronized next time data was to be received.

To solve this problem, it was initially attempted to make a form of "handshake" code. This code worked by first sending synch messages until an ack was received. Then the main message would be sent before controlling that the correct checksum was returned by the receiver. If not, the sending module would go back to sending the synch messages, and all this would repeat until the message was properly received. This did look like it was working, but was considered a rather bad solution to something that was obviously not working right.

It was later understood that the problem appeared as the port would keep being reinitialized to using a baud rate of 115200. It was then understood that this happened as Angstrom would run getty on the port. This would lead to the port continually resetting, and therefore not being able to work with the XBees. The problem was then finally solved by commenting out this line `"S:2345:respawn:/sbin/getty 115200 ttyS2"` in \etc\inittab:

Then the BeagleBoard would be restarted to activate the change. It was also mentioned on some forums, that if the problem did not get solved from doing this, you had to remove the phrase in bootargs where you specifyed "console = ttyS2, 115200". This was not done, as the problem in this case was fixed only by removing the line in \etc\inittab.

6.2.2.2 Player module

For the player modules, the XBees would be connected to the UARTs of the AVR controllers. The first prototype (chapter 5.2.1) used E-Blocks that ran on 5V. As the XBees run on 3.3V, a similar converter to the one used for the main module, was also used for this prototype.

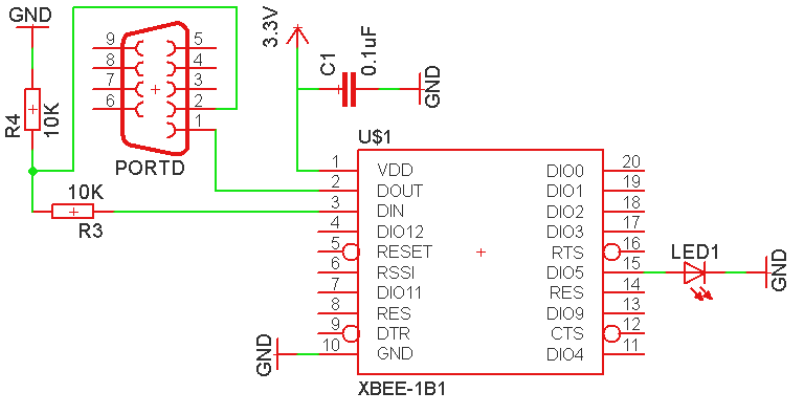


Figure 6-8: Schematics connecting XBee module to player module, prototype 1

The second prototype (chapter 5.2.2) was already operating at 3.3V, so no conversion was necessary and the XBee could be connected directly to the UART via one of the expansion headers on the A3BU.

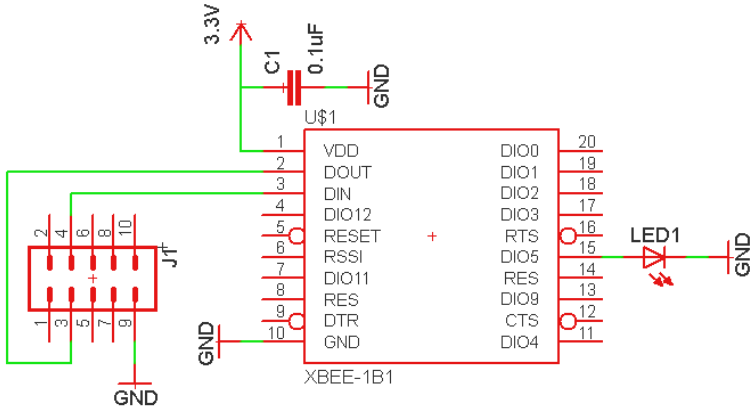


Figure 6-9: Schematics connecting XBee module to player module, prototype 2

UART setup

When making the first prototype, no premade libraries were used. The USART was then initialized manually using the following code:

```
unsigned int baudrate = BAUD_RATE_9600;

/* Enable interrupts */
sei();

/* Set the baud rate */
UBRR0H = (unsigned char) ( baudrate >> 8 );
UBRR0L = (unsigned char) baudrate;

/* Enable UART receiver and transmitter */
UCSR0B = ( ( 1 << RXCIE0 ) | ( 1 << RXEN0 ) | ( 1 << TXEN0 ) );

/* Set frame format: 8 data 1 stop */
UCSR0C = ( 3 << UCSZ00 );
```

Prototype 2 on the other hand, would use premade libraries. These were added through the *AVR Software Framework Wizard* when generating the application by adding the USART module to the project.

When this was done, the file *conf_usart_serial.h* was configured to the following:

```
#define USART_SERIAL_XBEE                &USARTC0
#define USART_SERIAL_XBEE_BAUDRATE       9600
#define USART_SERIAL_CHAR_LENGTH         USART_CHSIZE_8BIT_gc
#define USART_SERIAL_PARITY              USART_PMODE_DISABLED_gc
#define USART_SERIAL_STOP_BIT            false
```

The USART could then be initialized like this:

```
/* Enable interrupts */
sei();

/* USART options */
static usart_rs232_options_t USART_SERIAL_OPTIONS = {
      .baudrate = USART_SERIAL_XBEE_BAUDRATE,
      .charlength = USART_SERIAL_CHAR_LENGTH,
      .paritytype = USART_SERIAL_PARITY,
      .stopbits = USART_SERIAL_STOP_BIT
};

/* Initialize usart driver in RS232 mode */
usart_init_rs232(USART_SERIAL_XBEE, &USART_SERIAL_OPTIONS);
usart_set_rx_interrupt_level(USART_SERIAL_XBEE, USART_INT_LVL_HI);
```

Sending and receiving

When the USARTs were initialized, the modules could send and receive data through the RX and TX pin on the USART. For prototype 1, sending a char was done like this:

```
void xBee_Transmit( char data )
{
	/* Wait for empty transmit buffer */
	while ( !( UCSR0A & (1<<UDRE0)) );
	/* Put data into buffer, sends the data */
	UDR0 = data;
}
```

While the second prototype would use the following function:

```
void xBee_Transmit( char data )
{
	usart_putchar(USART_SERIAL_XBEE, data);
}
```

Receiving on the other hand, was done using interrupts to handle incoming data.

```
/* Interrupt handler */
ISR( USARTC0_RXC_vect )
{
	unsigned char data;
	unsigned char tmphead;
	ioport_set_pin_low(LED1_GPIO);

	/* Read the received data */
	data = usart_getchar(USART_SERIAL_XBEE);
	/* Calculate buffer index */
	tmphead = ( USART_RxHead + 1 ) & USART_RX_BUFFER_MASK;
	USART_RxHead = tmphead;	/* Store new index */

	if ( tmphead == USART_RxTail )
		/* ERROR! Receive buffer overflow */
		ioport_set_pin_high(LED0_GPIO);

	USART_RxBuf[tmphead] = data; /* Store received data in buffer
*/
}
```

The code above is from prototype 2, but the only difference between the two prototypes was the name of the interrupt vector (USART0_RX_vect vs. USARTC0_RXC_vect), the function for receiving a char (data = UDR0 vs. data = usart_getchar(USART_SERIAL_XBEE)) and the way status LEDs are lid. It was also necessary to enable high level interrupts on the A3BU by adding the following line:

```
PMIC.CTRL = PMIC_HILVLEN_bm;
```

The read functions for the two prototypes, would then look exactly the same, except for how the status LED was lid.

```
/* Read and write functions */
unsigned char xBee_Read( void )
{
	unsigned char tmptail;

	if ( USART_RxHead == USART_RxTail )  /* No data in receive
buffer */
		return 0x00;

	/* Calculate buffer index */
	tmptail = ( USART_RxTail + 1 ) & USART_RX_BUFFER_MASK;
	USART_RxTail = tmptail;          /* Store new index */

	if(!xBee_DataInReceiveBuffer())
		ioport_set_pin_high(LED1_GPIO);

	return USART_RxBuf[tmptail];     /* Return data */
}
```

All other functions used when handling the XBees, would look the same for both player modules. They are all added to the digital attachment under *BeagleBoard – Plaer modules – A3BU:* and – *E-Blocks*: *header – xBee.h* and *src – xBee.c*.

6.3 Separating messages

When sending several messages, they may get heaped up while being processed by the player modules. To separate each message, they were always ended by an 'X'. Reading messages would then be done the following way.

```
while (xBee_DataInReceiveBuffer()){
	newChar = xBee_Read();
	if( newChar == 'X' )
		break;
	data[i++] = newChar;
}
```

In hindsight it would probably have been better to use '/0' for separating the messages.

6.3.1 Addressing and command mode

Addressing

When playing Texas Hold'em, the main module will sometimes need to send messages to specific player modules. An example is at the end of a round, when the pot should be added only to the bankroll of the winning player(s).

Every RF data packet sent over-the-air contains a Source Address and Destination Address field in its header. The RF module conforms to the 802.15.4 specification and supports both short 16-bit addresses and long 64-bit addresses. A unique 64-bit IEEE source address is assigned at the factory and can be read with the SL (Serial Number Low) and SH (Serial Number High) commands. Short addressing must be configured manually. A module will use its unique 64-bit address as its Source Address if it's MY (16-bit Source Address) value is "0xFFFF" or "0xFFFE".

To send a packet to a specific module using 64-bit addressing: Set the Destination Address (DL + DH) of the sender to match the Source Address (SL + SH) of the intended destination module [22].

Command mode

To enable addressing specific modules, the XBees would need to be set into command mode. In this state, incoming characters will be interpreted as commands to the module instead of messages that will be transmitted to other modules.

The default sequence for transition to command mode, is as follows:

- No characters sent for one second [GT (Guard Times) parameter = 0x3E8]
- Input three plus characters ("+++") within one second [CC (Command Sequence Character) parameter = 0x2B.]
- No characters sent for one second [GT (Guard Times) parameter = 0x3E8]

When in command mode, AT commands can be sent using the following syntax:

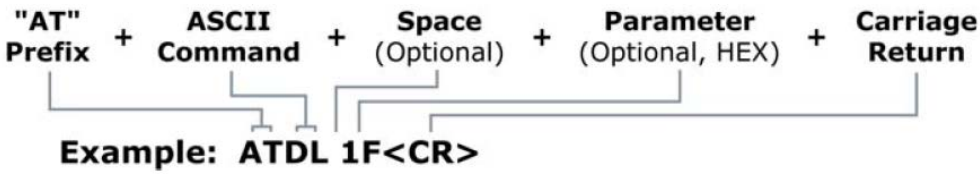


Figure 6-10: Syntax for sending AT Commands

After having sent a command, the XBee will return an "OK" message unless the command results in an error, in which case it returns an "ERROR" message. When the wanted commands have been sent, command mode can be excited by issuing an ATCN command to the XBee.

In this project, the following commands have been used.

AT Command	Command Category	Name and Description	Parameter Range	Default
DH	Networking {Addressing}	**Destination Address High**. Set/Read the upper 32 bits of the 64-bit destination address. When combined with DL, it defines the destination address used for transmission. To transmit using a 16-bit address, set DH parameter to zero and DL less than 0xFFFF. 0x000000000000FFFF is the broadcast address for the PAN.	0 - 0xFFFFFFFF	0
DL	Networking {Addressing}	**Destination Address Low**. Set/Read the lower 32 bits of the 64-bit destination address. When combined with DH, DL defines the destination address used for transmission. To transmit using a 16-bit address, set DH parameter to zero and DL less than 0xFFFF. 0x000000000000FFFF is the broadcast address for the PAN.	0 - 0xFFFFFFFF	0
MY	Networking {Addressing}	**16-bit Source Address.** Set/Read the RF module 16-bit source address. Set MY = 0xFFFF to disable reception of packets with 16-bit addresses. 64-bit source address (serial number) and broadcast address (0x000000000000FFFF) is always enabled.	0 - 0xFFFF	0
SH	Networking {Addressing}	**Serial Number High**. Read high 32 bits of the RF module's unique IEEE 64-bit address. 64-bit source address is always enabled.	0 - 0xFFFFFFFF [read-only]	Factory-set
SL	Networking {Addressing}	**Serial Number Low**. Read low 32 bits of the RF module's unique IEEE 64-bit address. 64-bit source address is always enabled.	0 - 0xFFFFFFFF [read-only]	Factory-set
CN	AT Command Mode Options	**Exit Command Mode**. Explicitly exit the module from AT Command Mode.	--	--
GT	AT Command Mode Options	**Guard Times**. Set required period of silence before and after the Command Sequence Characters of the AT Command Mode Sequence (GT+ CC + GT). The period of silence is used to prevent inadvertent entrance into AT Command Mode.	2 - 0x0CE4 [x 1 ms]	0x3E8 (1000d)

It was also planned to use sleep commands to reduce power consumption. Different networking commands would then be used to configure the sleep times as well as defining PAN ID and channel flag (see chapter 2, XBee®/XBee-PRO® Networks in [22]). As this was not crucial for the functionality of the system, it was not prioritized to improve this further at this time.

6.3.2 Transparent Operation

When powering up the XBee modules, they will be operating in Transparent Mode. In this mode, the modules act as a serial line replacement. All UART data received through the DI pin will be queued up for RF transmission. When RF data is received, the data will be sent out the DO pin.

6.3.2.1 Testing the modules

Before starting setting the XBees up using command mode, the default transparent operation was used to see that the interface with the main module and player module was working as expected.

This was done in two steps. First the XBee modules them self were tested by connecting them to the COM port on the stationary computer used during software developing. COM ports had already been used in this environment so that would remove some sources of error during testing.

For the actual testing, a simple windows form was made that would connect to a given COM port. It would then print all data received on the port to a textbox. It could also send data to the port.

On the ATmega, a program was made that would send data on the UART, corresponding to what button was clicked. It would also print all received data on to the led display.

To do this, no configuring needed to be done to the XBees. As long as the logic levels were converted as described in the above chapter, and the TX and RX cables were connected correctly, communication would work as soon as the modules were powered up.

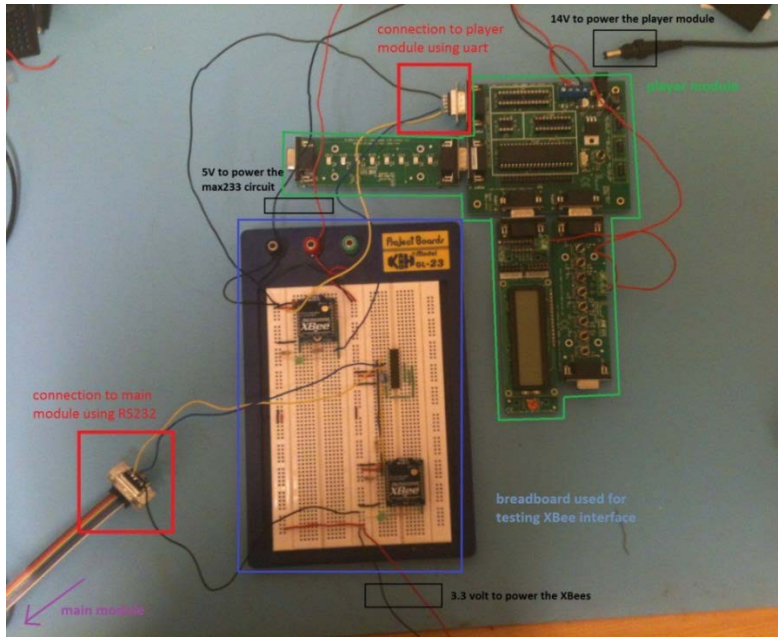


Figure 6-11: Setup used when testing the XBee modules

This test proved that all the XBee modules worked in transparent mode and that the USART on the player module was configured correctly.

When the test was completed, the main module XBee could be connected to the BeagleBoard instead of the stationary computer. Transparent mode could then be used when figuring out how to configure the RS232 port on the Beagle. The resulting procedure found from doing this, was explained in chapter 6.2.2.1.

6.3.3 Unicast mode

When playing Texas Hold'em, a lot of data will be broadcasted (updates on pot and high bet, cards read and other general information). Still, some data will only be sent to specified players. This includes activating a player module so it can send info on what action is taken by the player, informing the player module if it is to work as small blind, big blind or dealer and it is used by all player modules, so that only the main module receives messages sent.

To send a packet to a specific module, the Destination Address (DL + DH) on the sender must match the Source Address (SL + SH) of the desired receiver. For this project, it was decided to use 64-bit addresses, as this would provide unique addresses for all player modules. This was achieved by setting the MY parameter on the XBees to 0xFFFF. The source address of the modules would then be set to the modules serial number.

To achieve point-to-multipoint communication, the main module would need the address of all the player modules. To get this information, functions were made to register player modules as active players.

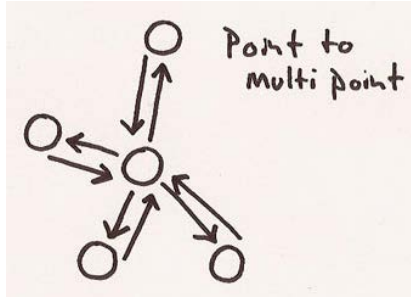


Figure 6-12: Illustration of point to multipoint communication, [24]

On the BeagleBoard, the player registering was done by broadcasting the source address of the Beagle every third second. Between broadcasting, it would see if any player modules had responded to the broadcast. If a player replied, it would be added to a list of registered players and then the player would be notified that he/she had been registered. If the same player module was to reply a second time, it would be detected as already registered, and therefore not be added a second time, but instead just be notified that the registration had been successful. Players would keep being registered until one of the modules reported that the game could begin.

The player modules on the other hand, would listen for the address of the main module for up to five seconds. If the address is received, the destination address of the module would be set to correspond to the address of the main module. This would prevent other player modules form receiving the messages sent, as that would complicate the functions for receiving messages.

When the player module has detected the main module and set the destination address, it will reply by sending its personal address. Then it will wait up to four seconds for a confirmation that the registration is complete. If no confirmation is received, the address will be resent. This procedure will be done up to five times, or until the confirmation is received.

The main functions used by both modules are added in appendix 13.2.4.1 and 13.2.4.2. All functions used are also added to the digital attachment.

6.3.4 Broadcast Mode

Broadcasting is used when a message is intended for all players. To activate broadcast mode, the destination address is set as follows:

```
DL (Destination Low Address) = 0x0000FFFF
DH (Destination High Address) = 0x00000000 (default value)
```

Then messages are sent as usual, only that they now will be received by all player modules in range.

6.3.5 Guard Times (GT)

One problem after setting up the point-to-multipoint communication was regarding Guard Times. GT was briefly mentioned above and is the time-of-silence that surrounds the command sequence used to enter command mode (CC Command). When entering command mode, the following sequence is used, GT + CC + GT.

The problem that appeared was that the default GT was one second (0x3EB). It would then take two seconds every time the module entered command mode. This happened quite often, as it is necessary when changing the destination address of the module and when activating/deactivating broadcasting.

During a bid round, the destination address will change every time the turn shifts to a new player. It will then also broadcast the new high bet to all players. When this took two seconds every time, it led to a noticeable delay. The delay also became quite noticeable when registering players before starting the actual game as well as when initializing players at the start of a new round.

To decrease the delays, the GT parameter was changed till 200 milliseconds (0xC8) during the initialization of the XBee. This proved to be fast enough for the delays mentioned to no longer be noticeable.

7 Playing Texas Hold'em

To play a game of Texas Hold'em, three main components were required. These included a deck of cards, the rules of the game and the players. How these were handled by the Texas Hold'em application is explained in this chapter.

The application was initially developed on a stationary computer. It was then cross compiled to run on a BeagleBoard. During development there was also made a small application to display the cards detected for the game as well as some game stats.

7.1 The Texas Hold'em application

The Texas Hold'em application made in this project would follow the game rules added in appendix 13.1. The application took raw data received from the computer vision program (chapter 3) and interpreted it regarding the current state of the game. Actions taken by human players were received by the main unit (chapter 4) from different player modules (chapter 5), while actions taken by James "himself" were automatically generated within the application.

7.1.1 Game structure

For readers unfamiliar to the game of Texas Hold'em, the rules are added in appendix 13.1. Here is a short summery that should be sufficient for further reading.

Figure 7-1: Texas Hold'em table with two aces as pocket cards

Each round has the following structure. Initially all players are dealt two cards each, which are known as the 'pocket cards'. Then there is a betting

round before the first three of a total of five community cards are dealt, known as the 'flop'. Then the game goes on like this; betting round, one more community card (the 'turn'), betting round, last community card (the 'river'), last betting round and finally the 'showdown' where the winner of that round is declared. In each betting round, a player can 'fold', 'call/check' or 'bid/raise'. When folding, the player drops out from the rest of the round, and can therefore not win the current round. If only one player is left before the showdown, he is pronounced the winner of that round and gets the money in the pot. When a player is out of money, he is no longer a part of the game. The game ends when only one player is left.

The main functions for playing the game include the following:

- startRound()

  This function will set all players as active and then clear their pocket cards. It will also clear all game cards detected from previous rounds. Then blinds (small initial amount of money) will be posted from the players currently acting as big and small blind. Finally player stats are sent to all player modules.

- bidRound(bool preFlop = false)

  The bidRound function will go around the table and get actions taken by players until all players have been able to bet and all active bets are matching. The talker is the player that currently is allowed to take an action (bet, fold, call etc.). The initial talker is the player following the dealer, unless the preFlop parameter is set as it then will be the player following the small blind.

  If James is the talker, the action taken will be decided by the application. Currently this is done by simply by always making him check/call, but the plan is to make this decision using the same principles as open source projects used to play online poker.

  At the end of the function, all bet are collected from the player's bankrolls and added to the pot. Finally, the new player stats are sent to all player modules.

- showdown()

  In the showdown, a value is calculated for the best five-card poker hand of each player. The function for calculating these values are described further in chapter 7.1.3, Hand evaluation.

- devidePot()

  Here the pot will be equally divided between the players with the best hands.
  After the pot is awarded, the position of the dealer is moved to the next player and updated player stats are transmitted to all player modules.

- Read Functions

  The read functions are where input from the computer vision program is interpreted. These are covered in more detail in a separate chapter below.

For a round of poker, these functions are structured the following way:

```
void TexasHoldem::playTexasHoldem()
{
      startRound();
      james->cvReadPocket( theBoard );

      bidRound(true );
      if(activePlayers( ) < 2)
            goto endRound;

      theBoard->cvReadFlop( );
      bidRound();
      if(activePlayers( ) < 2)
            goto endRound;

      theBoard->cvReadTurn( );
      bidRound();
      if(activePlayers( ) < 2)
            goto endRound;

      theBoard->cvReadRiver( );
      bidRound();

endRound:
      showdown();
      devidePot( );
}
```

This procedure will then repeat over and over until the game ends.

7.1.1.1 Missing functionality

Due to lack of time, not all functionality for playing a full game of Texas Hold'em was implemented. Among the parts still missing, was:

- Handling when players went out of chips, and therefore had to leave the table.
- Allowing for new players to join an ongoing game.
- Open up for different rule variations (see below)
- Dividing the pot correctly if the winner had gone all in while other players had bet more than this amount.
- Players folding during showdown
- Saving games

7.1.1.2 Rule variations

In the appendix, Texas Hold'em rules have been added. Here it is seen how this game can be played in a few different variations. The rules will mostly be the same for each of these with a few exceptions like how much a player is allowed to bet in a single round. The variations are as follows; limit, no limit, pot limit and mixed Texas Hold'em, and are all explained in appendix 0.

The game that was implemented in this project was the no limit version. The plan was that when starting a new game, players could choose what version of the game they wanted to play (as well as being able to choose entirely different games than poker).

7.1.2 Read functions

The computer vision program would return a string containing info on all cards detected in a single frame from a web camera. This string includes the rank and suit of the cards, a value indicating how exact the rank and suit identifications were as well as the position of the cards.

Figure 7-2: Example output from the computer vision program

In the example from Figure 7-2, the cards jack of hearts, three of clubs and four of spades were laying on a table. The entire card was visible for the jack and three, while only half the card was visible for the four.

When playing Texas Hold'em, up to five cards may lie on the table and the players may have two pocket cards each (unless they have folded). The five cards on the table are being dealt in three rounds. First three cards called the flop, then one card called the turn and finally a card called the river. The pocket cards belonging to each player will only be read if the player participates in the showdown at the end of each round.

As the computer vision program may wrongly identify cards, miss a card or falsely detect a card that is not really on the table, it is not sufficient to only get the results from a single frame. Instead, a function was made that would use the results from several subsequent frames to decide on what cards were actually laying on the table.

This function would take three parameters. The first parameter is an array of cards. These will be the cards that the function is attempting to read. The next parameter is the number of cards expected to be read. The final parameter is a pointer to the game board. This board contains a vector called deadCards that contains all cards that already has been detected. This being cards on the table or cards belonging to players. The vector is used so that cards on the table will not be detected twice. The board parameter was initially used for more reasons than accessing the deadCards vector. The way the function works now, it would probably be a better solution to only include a reference to this vector instead of the entire board object.

Reading the flop, turn and river was done within the Board class, and the readCards function would be used the following way for the flop:

```
while(!flopIsFound())
      readCards(flop, 3, this);
```

For the turn it would look like this, as the turn card was a single card instead of an array of cards:

```
while(!turnIsFound())
      readCards(&turn, 1, this);
```

Reading pocket cards on the other hand, was done within the Player class. Here the main game board was included as a reference called theBoard. The function would then be used like this:

```
while(!pocketIsFound ())
      readCards(pocket, 2, theBoard);
```

The readCards function is the part of the Texas Hold'em application that uses the computer vision function. When having retrieved card information from a frame, the function will update the array of cards to be found. It the same card is detected in more the one frame, the strength of this card is increased. If a previously detected card is not re detected in another frame, the strength of this card will decrease. The pseudo code for the readCards function is given below.

```
readCards( BoardCard oldCards[], unsigned short nrOfCards, Board
*theBoard )
    get newCards from camera frame
    remove deadCards from newCards
    if no new cards are found
        decrease strength of all old cards
    if more cards are detected than expected
        only take care of the nrOfCards strongest new cards
    for all oldCards
        if a new card has same rank and suit as the old card
            increase strength of old card
            mark old card as updated
            erase card from newCards
    for all oldCards not yet updated
        if a new card is close to the old card
            if new card and old card has same rank
                increase rank strength of old card
            else
                decrease rank strength of old card
                if rank of new card stronger than rank of old card
                    set rank of old card = rank of new card
            if new card and old card has same suit
                increase suit strength of old card
            else
                decrease suit strength of old card
                if suit of new card stronger than suit of old card
                    set suit of old card = suit of new card
            mark old card as updated
            erase card from newCards
    while still more new cards
        get weakest old card not yet updated
        decrease rank strength of old card
        decrease suit strength of old card
        if total strength of old card < total strength of new card
        set old card = new card
        mark old card as updated
        erase card from newCards
```

The entire code can be found in appendix 13.2.5, as well as in the following to the digital attachment under *BeagleBoard – Main unit: cardGame.cpp*.

7.1.3 Hand evaluation

The hand evaluation function would return a value representing the strongest possible 5 card poker hand from an array of seven cards. For the Texas Hold'em application, this function was used during the showdown. The seven cards would then consist of the five community cards plus the two pocket cards for each player.

The function worked by testing every poker hand rank (see appendix 13.1.3) starting with the one with highest value (straight flush) down to the one with lowest value (high card). When a hand rank had been found, a number was generated to determine the strength of this hand. The code used for doing this is added in appendix 13.2.6, as well as in the to the digital attachment under source code, *BeagleBoard – Main unit: handValue.h* and *handValue.cpp*.

7.1.4 Getting player actions

In Texas Hold'em, player actions are taken during bid rounds in the game. As James the Poker bot would be playing against human players, this information would be received from external player modules (see chapter 5).

Each player would be allowed to determine what action to take when the player is set as talker. When James "himself" was the talker, the action taken would be determined within the Texas Hold'em application. Player actions were acquired the following way.

```
// in bidRound()
      (…)
      if( talker == jamesPos )
            james->makeCoise( highBet ); // currently only checks
      else
            players[talker]->getAction( highBet, comDevice);
      (…)

void Player::getAction(int highBet, xBee *comDevice )
{
      char command[COMMAND_SIZE] = "\0";

      // informs player module that that it is the current "talker"
      theGame->sendTo( playerNr, "T");
      comDevice->readMsg(command, true, 0, ACTION_T);

      switch (command[0]){
      case 'F': // Fold
            clearPocket();
            active = false;
            break;
      case 'C': // Call/Check
            bet = highBet;
            break;
      case 'R': // Bet/Raise
            bet = atoi(command+1);
            break;
      }
}
```

As commented in the code above, the makeChoise function (used to determine the actions taken by James) will currently only check. This was done as no functionality was made to calculate the probability of James's current hand winning the round. They way this was planned to be solved, is further described in chapter 10, Discussion.

7.2 Windows debug program

When developing the Texas Hold'em application, a small windows form application was made to get a visual on how computer vision results were interpreted. The form would be connected to a serial port and every time a card was identified using one of the read functions, the results would be posted on this port.

There also was an idea that this program could be activated by connecting the final main module to the COM port of an external computer. The program would then be more extensive, and would be used to interfere with the robot.

7.2.1 The tools

The program was designed as a *Windows Form Application* using visual studio 2010 and consisted of the following tools:

- 7 Windows::Forms::PictureBox
- 1 Windows::Forms::ImageList
- 1 IO::Ports::SerialPort
- 2 Windows::Forms::Button
- 1 Windows::Forms::Label

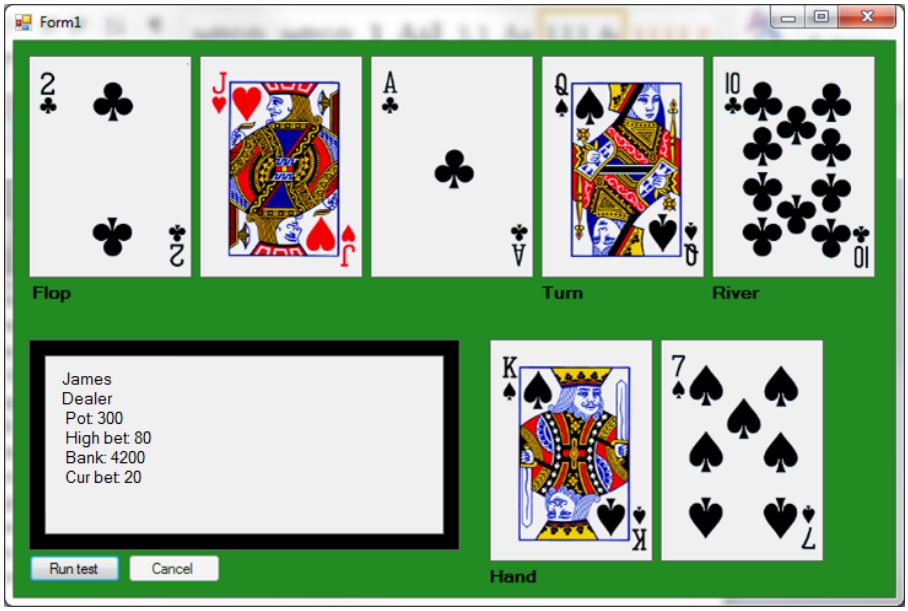


Figure 7-3: Window generated by the debug application

The way these tools interacted is described in the following sub chapters.

7.2.1.1 Picture boxes

The picture boxes are used to display images of the detected playing cards.

7.2.1.2 Image list

The image list is used to store card images that could be displayed in the picture boxes.

7.2.1.3 Serial port

The serial port is used to communicate with the Texas Hold'em function. This is where information would arrive regarding what cards had been detected. When this happened, it would trigger an event. The code that then would be executed is added in appendix 13.2.7 as well as in the following digital attachment.

7.2.1.4 Buttons

The buttons are currently not working, but was planned to be used to interfere with the Texas Hold'em application.

7.2.1.5 Label

The label would display various information received on the serial port.

8 THE ROBOTIC ARM

For this project, the goal was that James would be able to play a game of Texas Hold'em. The game was chosen for the reason that as long as James do not act as a dealer, he would only have to handle two private cards.

8.1 For use with the Texas Hold'em application

If there had been enough time to complete a prototype of the arm during this project, the plan was to make it as simple as possible. The arm would fetch cards one by one, moved them to a position where they could be read by a secondary camera and then placed to the side of the table where they would not be in the way for the rest of the game. If James was to participate in the showdown, he should be able to display the cards to the opposing players and at the end of the round, the cards should be moved to about the middle of the table so that the dealer could pick them up.

8.1.1 Movement

Regarding the positioning of the mechanism used to actually pick up the cards, there were a few ideas. As mentioned, it was desirable to keep the arm as simple as possible. The first plan for moving the arm was using polar coordinates. The "card picker" would be placed at the end of a beam, where the beam was working as the arm itself. The angle would be set by rotating the arm at the point where the arm is connected to the table. The arm itself would also be able to slide back and forwards through this connection device to adjust the radius. Figure 8-1 demonstrates how this arm would be used to pick up a card and move it to a position where a secondary camera could identify it.

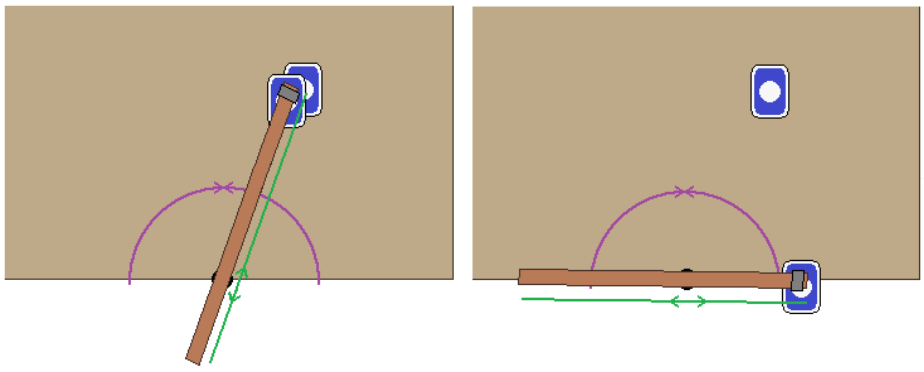


Figure 8-1: Robotic arm: idea nr 1.

It was considered whether initial idea would have a problem with the arm slanting downwards. This would especially be possibility if the "card picker" at the end of the arm was too heavy. Another idea that would make the arm more stable was to have to beams at each side of the expected area for where the private cards could be placed. The arm would then move using Cartesian coordinates. Here the y-axis would be controlled moving a third beam along the two side bars. To adjust the x-axis, the card picker would slide along this third beam. Figure 8-2 demonstrates how this arm would be used to pick up a card and move it to a position where a secondary camera could identify it.

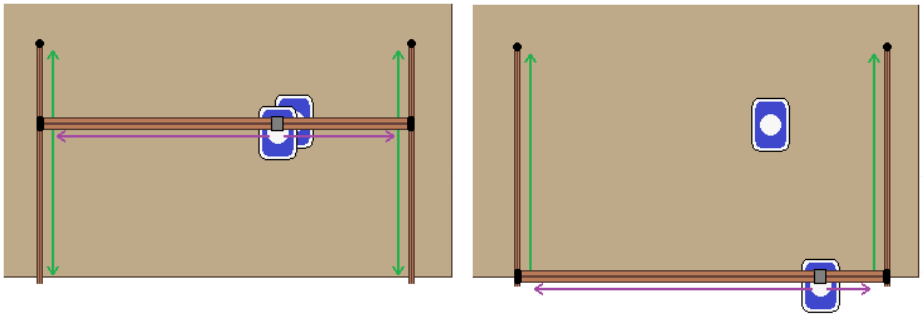


Figure 8-2: Robotic arm: idea nr 2

This idea would provide a very unwanted consequence with the side beams of the arm being "in the way" during the game.

The final idea for making the prototype robotic arm was sort of a hybrid of the first two ideas. Like idea nr one, it would move in polar coordinates. Instead of sliding the entire arm back and forwards, it would use the aspect form idea nr two and instead slide the "card picker" along the arm. Figure 8-3 demonstrates how this arm would be used to pick up a card and move it to a position where a secondary camera could identify it.

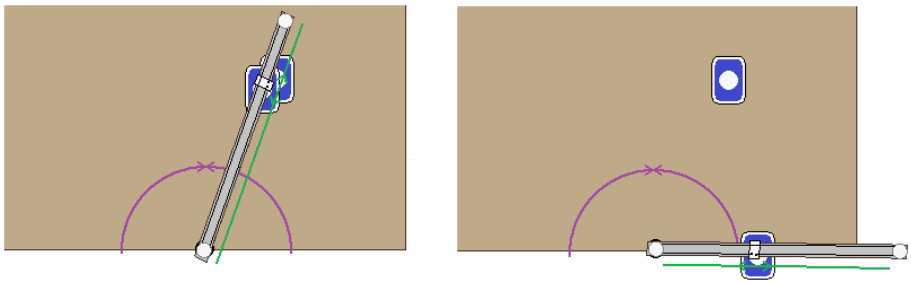


Figure 8-3: Robotic arm: idea nr 3, final idea used for prototype

A physical prototype was made of the third idea. This prototype could be prone to the same problem with the arm slanting downwards, as described for idea nr 1. The expected max weight of a potential "card picker" was added to the end of the arm, by manually pressing the arm downwards. From this simple test, it was assumed that slanting would not be a problem for this prototype.

8.1.1.1 Calibrating movement

To calibrate the arm position to correspond with the locations achieved from the card recognizing program, there could be a mark on the part of the arm used to pick up cards. A simple computer vision program could then be made to track this mark. Parameters for moving the arm could then continually be adapted so that the movements would be as expected. Another idea was that these parameters would be set at the beginning of the game in a separate configuration procedure, but this would require that the camera and arm mechanics would remain static for the rest of the game. A hybrid of these two solutions would also be a possibility.

8.1.2 Picking up cards

Another problem for making the robotic arm was how to actually pick up the cards. The prototype described above, would be able to move a potential "card picker" to a position above the card that would be picked up. A human would normally pick up a playing card by dragging the card to the edge of the table the card was placed on, before using a thumb to actually grab the card.

If James the PokerBot were to pick up cards like a human, this would demand a rather complicated mechanism. Instead it was planned to use air suction to lift the cards. A couple ideas on how this could have been done are explained below.

The first idea consisted of lowering a small plunger on to the card. Then a slight "vacuum" could be added by attaching a tube, connected to a piston, for then to lower the piston. To release the card, the piston could be pushed up again so the "vacuum" is removed and the remaining air pushes the card free from the plunger.

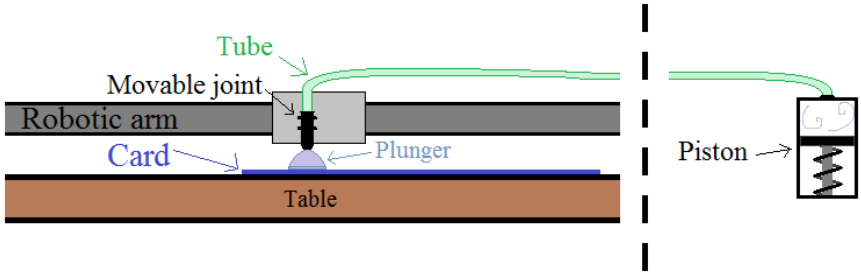


Figure 8-4: Card picker using plunger and piston

A few problems with this design, is that the cards will need to be completely straight where the plunger touches the surface, to get an airtight seal. Also the mechanism to lower the plunger on to the card may prove to be a bit complex.

Another idea was to connect the tube to a pump that could provide a constant suction. Then the tube could be attached to a static end piece that would hover just above the cards surface. Suction from the pump was then expected to be able to lift the card from the table, up to the end piece, and keep it there as long as the pump was active.

It is unsure how strong the suction from the pump would have to be to lift the card. This would depend on the distance between the end piece and the card, as well as the dimensions of the tube. It is also considered that this solution could be a bit noisy when moving the cards, depending on the pump.

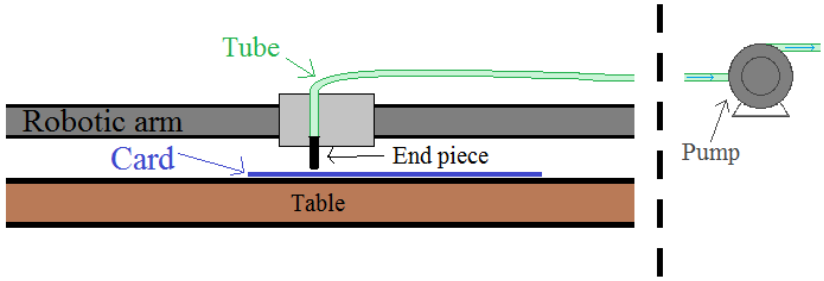


Figure 8-5: Card picker using pump

8.1.3 Showing card

At the end of the round, during the showdown, players need to show their private cards to determine the winner of that round. It should be mentioned that not too much thought has yet been put into possible solutions for this, but one plan was using the "card picker" to display the cards.

This could be done by simply lifting the card picker and then tilting it 45 degrees so that the card would be displayed for the other players to see. This mechanism would however require a rather precise positioning of the card picker relative to the card. Too long in on the card, and the cards edge would hit the table and therefore possibly be separated from the card picker when turning.

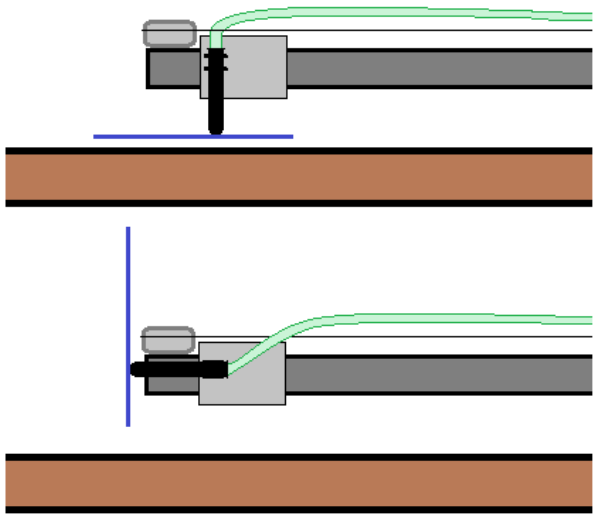


Figure 8-6: Displaying card

8.2 Picking up poker chips

Another aspect of a normal game of Texas Hold'em is using poker chips for betting. The current prototype will get bets using the player modules instead of poker chips. Still, it is planned that if work would be continued on the project, playing with normal poker chips would be a possibility. This would require additional functionality for the robotic arm.

The plan was that a new computer vision program would be made to localize and read poker chips (value and amount of chips in a stack). The robotic arm would then have to be able to move chips from James's current stacks to the middle of the table (when betting) as well as picking up chips from the pot and restacking them on to the correct stacks (when winning).

To move the chips, it was planned to add a "chip picker" to the "card picker" on the current arm. How too actually pick up the chips themselves had not yet been considered in any detail. Regarding moving the chips, it was considered doing this one by one on moving entire stacks. When moving the chips one by one, the "chip picker" could be on a vertical track of some sort. It could then lift a chip to a height corresponding to a max stack height and place the chips onto current stacks like in Figure 8-7.

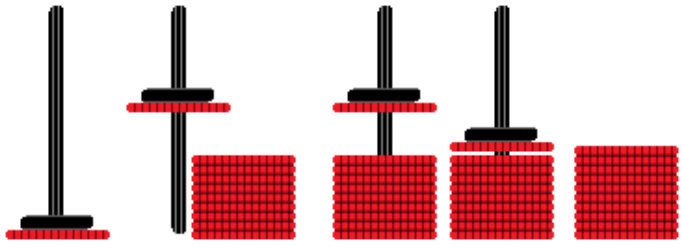


Figure 8-7: Moving chips one by one

Moving entire stacks could be done by locally stacking the chips above the "chip picker". This could be done by placing the stack on the opposite side of the track mentioned above as the actual "chip picker". The "chip picker" could then move the chip to the top of the track, spin the chip around to the side with the stack, add the chip to the stack and move the "chip picker" back to its original position. The local stack could then be added to a current stack as demonstrated in Figure 8-8.

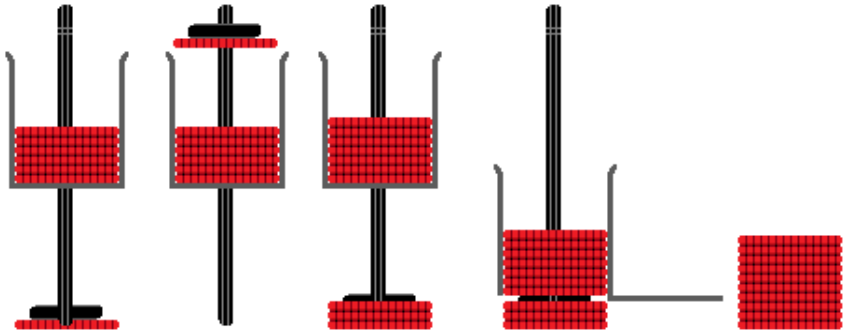


Figure 8-8: Moving entire stacks of chips

As the prototype arm will not be able to move above currently stacked chips, the stacks would have to be placed as in the following figue.

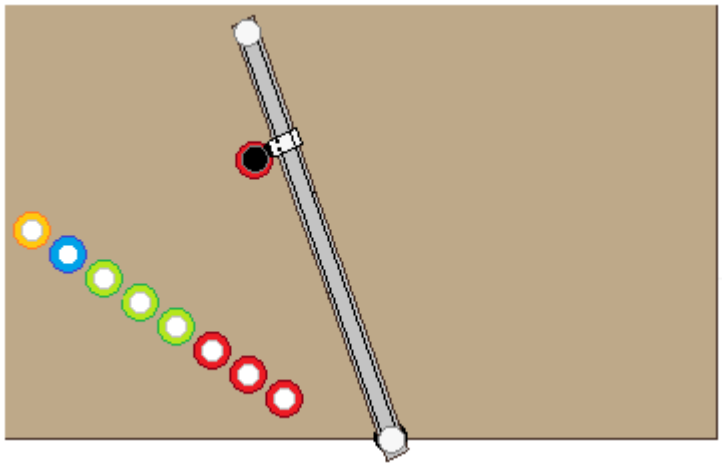


Figure 8-9: Placing stacks of chips

8.3 Other card games

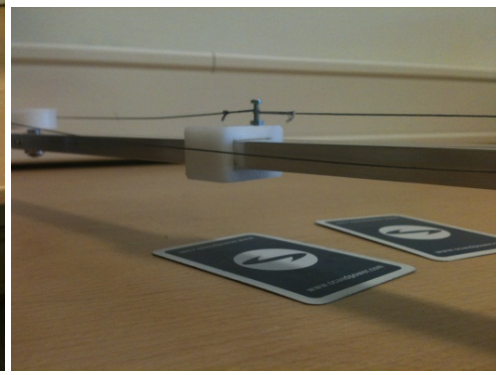
The robotic arm described above, is custom made to play a game of Texas Hold'em. This arm should be improved to also be able to play other varieties of card games. This would include card games where cards will be added and removed from a hand of several cards repeatedly during the game.

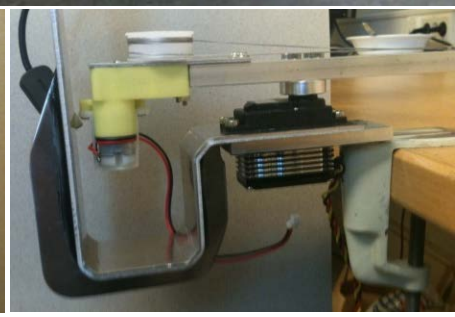
With the current arm, this could be achieved by simply having two stacks of cards representing the hand of cards. To fetch a specific card, the arm would simply move cards from one stack to the other until reaching the required card. Cards would then only have to be read before being added to one of the stacks. After this, the robot would have full control of where in the stacks each card is located.

Still the arm would demand a lot of extra mechanisms to be able to play any kind of card game. One problem would be placing cards faced up to specific positions on the table.

8.4 The prototype

A prototype of the mechanics for the robotic arm was made at the end of this project. Unfortunately, there was not enough time to program the arm to work with the Texas Hold'em application, or to make the mechanism for actually picking up the cards. Pictures are provided below.





9 Result

It was decided that the best way to demonstrate the results of this project was creating videos demonstrating the actual use of the final products. To do this, three videos were created. The first video shows the raw output from the computer vision program. The other two demonstrates how the final system worked, both when it was running on the stationary computer and when it was running on the BeagleBoard.

All videos are included in the digital attachment under *Result videos* as well as being available online at youtube.com under the following addresses.

- Computer vision result
  http://youtu.be/MHM0-zkwNsg
- Playing Texas Hold'em on the stationary computer
  http://youtu.be/vYG2iN8HISE
- Playing Texas Hold'em on the BeagleBoard
  http://youtu.be/J2RSRdu6jtU

For convenience when reading this thesis, some selected screenshots from each video along with an explaining text is given below. This is not intended as a replacement of the videos but rather meant to provide a summary of the contents of each video.

9.1 Computer vision result

These screenshots are the same as those provided in the separate result chapter (chapter 3.6) for the computer vision program.

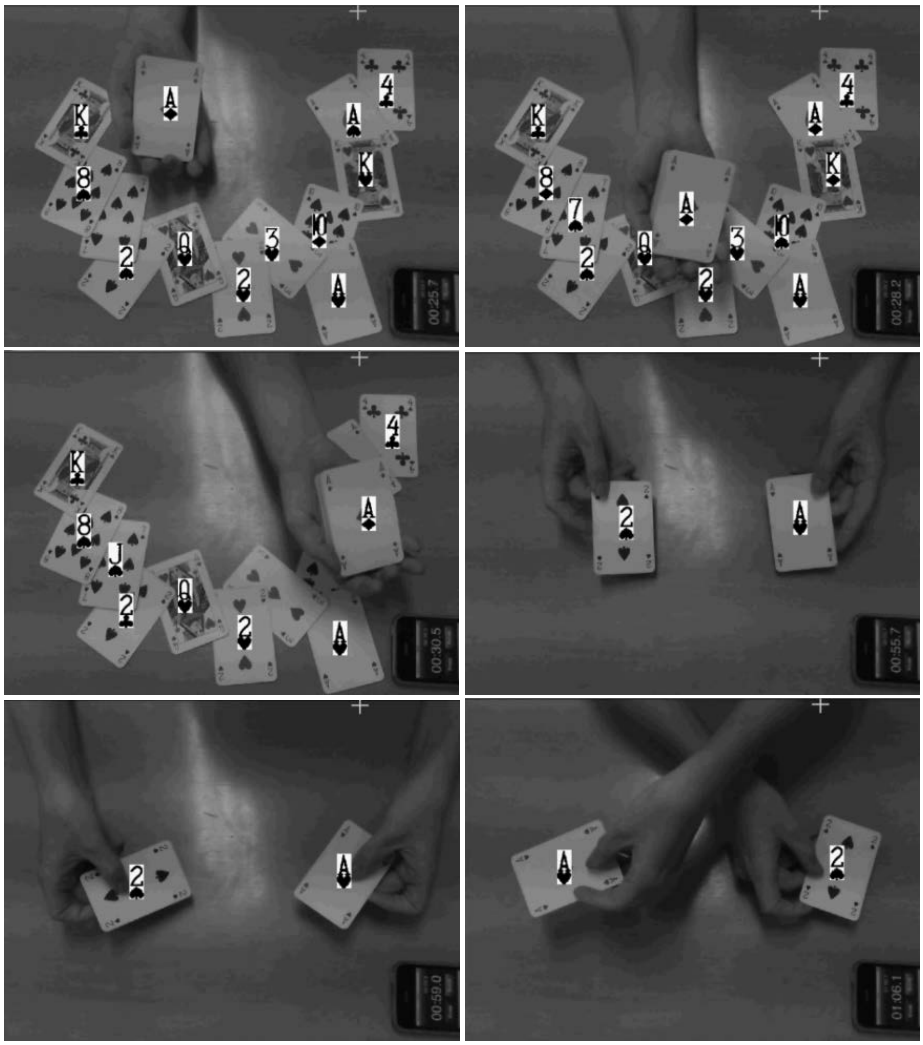
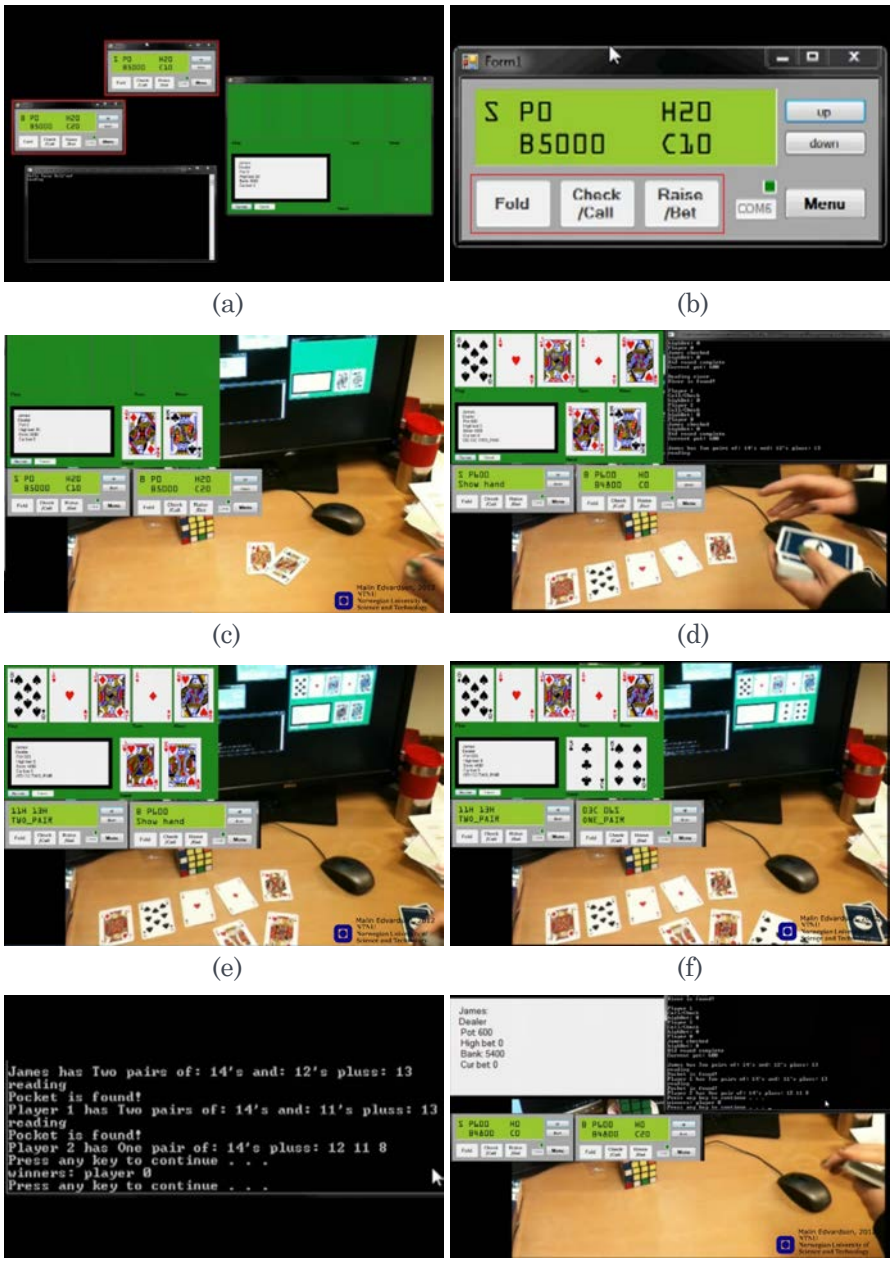


Figure 9-1: Raw result from the computer vision program

9.2 The Texas Hold'em application

This video demonstrates how the computer vision program could be used to play a game of Texas Hold'em. The following screenshots and explaining text are meant as a summary of the video.



(a) (b)

(c) (d)

(e) (f)

(g) (h)

Figure 9-2: Screenshots from video demonstrating the Texas Hold'em application

a) The video starts by explaining the different windows that will appear on screen when running the application. These include the terminal window that provides output from the application itself, the player module simulators and the debug program used to print the cards detected for the game. Red squares will surround each window as it is explained.
b) Then the player module simulators are explained a bit further, also this time using red squares to highlight the areas that are discussed.
c) When all the output on the screen is explained, the game starts. First the pocket hand belonging to James is dealt. It is then seen how the computer vision program detects these cards and print them to the screen.
d) Then all the community cards are dealt with bid rounds in between. When the last bid round is complete the video will pause and a closer look is taken on the current game stats.
e) After having reviewed the game stats, the showdown continues with player 1 showing his pocket cards.
f) The same is then done for player 2.
g) When all players pocket cards have been read and evaluated, the winner is determined and the result shown in the terminal window.
h) It is then taken a look at the player's game stats to see that only the winner has received the pot.

9.2.1 Response times

In this video, almost all the cards are detected by the computer vision program and correctly identified corresponding to the game within 2 seconds. The exceptions are the queen of heart (river) and the jack of heart (player 1's pocket card) that both take about 5 seconds to identify. This was assumed to be a result of a badly generated heart template. It was not attempted to improve this.

9.3 Playing Texas Hold'em on the BeagleBoard

This video demonstrates how the computer vision program and Texas Hold'em application can be run on a BeagleBoard. Human player will here use physical player modules, communicating with the BeagleBoard over ZigBee, to participate in the game. To film this video, three cameras were used. One was used to film the big screen, the web camera used for the computer vision and the cards that are put on the table while the other two were used to film the player modules.

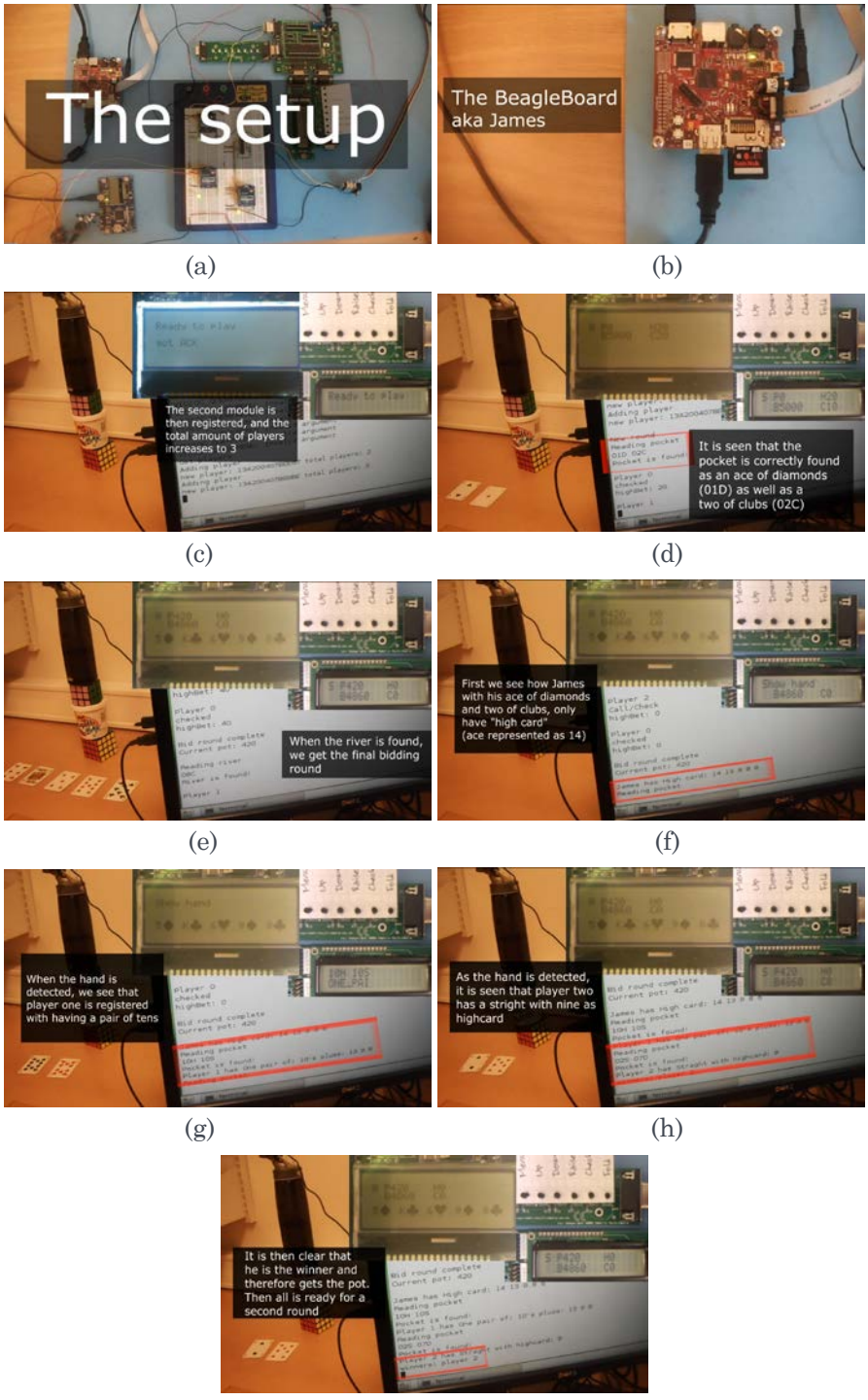


Figure 9-3: Screenshots from video demonstrating the Texas Hold'em application

a) The video starts by introducing the different modules used to play the game.
b) Each module as well as how it is connected to the corresponding XBee on the bread board, is here filmed close up.
c) Before the game starts, both player modules are registered at the main module.
d) When the game starts, James gets his pocket cards and we see how they are correctly identified by the computer vision program. The cards are presented to James manually, as the robotic arm never was completed.
e) Then all the community cards are dealt with bid rounds in between. When community cards are dealt, the second player module will print what is seen by the computer vision program.
f) After the last bid round, it is time for the showdown. Fist it is shown how James is correctly registered with only having a "High card" hand, ace top.
g) Next, player nr 1 is told to show his hand. The hand is read and evaluated, and hi is registered with having a pair of tens.
h) Finally player nr 2 shows his hand. As he has a seven and there already is a five, six, eight and nine on the table, he is correctly registered with having a straight.
i) It is then clear that he is the winner and therefore gets the pot. Then all is ready for a second round.

9.3.1 Response times

In this video, the response time for identifying the different cards is as follows:

- Pocket cards, James: 6 seconds
- Flop: 11 seconds
- Turn: 6 seconds
- River: 4 seconds
- Pocket cards, player 1: 9 seconds
- Pocket cards, player 2: 6 seconds

9.4 Digital Attachment

The digital attachment is organized the following way.

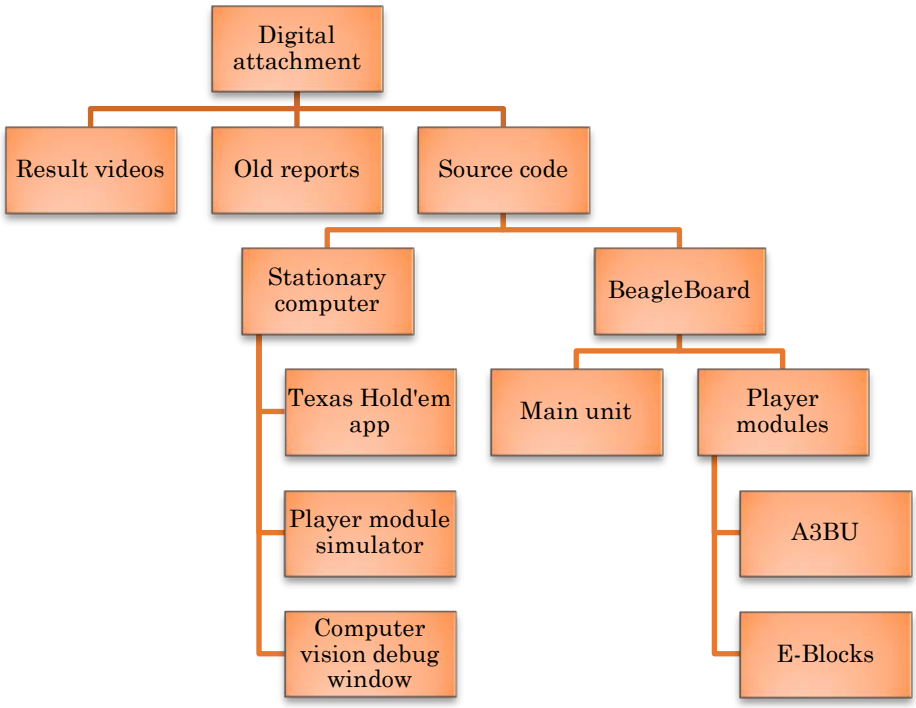


Figure 9-4: Organisation of the digital attachmment

Result videos contain the videos explained in the above sub chapters. These videos can also be found online.

Old reports contains the report for the original computer vision project made in the computer vision course, spring 2011 as well as the more advanced project made for the computer vision, fall 2011.

9.4.1 Compiling the source code

For the stationary computer, Microsoft Visual Studio 2010 was used as IDE. Three projects were made. One *Visual C++ Empty Project* for the Texas Hold'em app and two *Visual C++ Windows Forms Applications* for the player module simulator and computer vision debug window.

OpenCV was installed on the computer, as this was used to get frames for a webcamera, and the following files were added to the Texas Hold'em project:

Include Directories:	*C:\OpenCV2.1\include\opencv*
Library Directories:	*C:\OpenCV2.1\lib*
Source Directories:	*C:\OpenCV2.1\src\cv* *C:\OpenCV2.1\src\cvaux* *C:\OpenCV2.1\src\cxcore* *C:\OpenCV2.1\src\highgui* *C:\OpenCV2.1\src\ml*

For the BeagleBoard, the procedure used for cross compiling the code, is explained in chapter 4.2.2. For the player modules, AVR Studio 5.1 was used as IDE.

Player module nr 1 (the one based on E-Blocks), had all its code made from scratch. The module was programmed using an AVRISP mkII as tool and ATmega324P as device.

Player module nr 2 on the other hand, used a lot of premade code. First the project was started as a *User application template – XMEGA-A3BU Xplained*. This would add the appropriate files for initializing the board. Then all files needed were added using AVR's *AVR Software Framework Wizard*. Here modules for controlling the display, USART, interrupts and IOports were added. The player module was programmed using a JTAGICE 3 from AVR as tool and ATxmega256A3BU as device.

10 Discussion

This discussion begins by taking a look at the working proses that have been used during this project as this is considered quite a large part of the project itself. Then the final results are discussed followed by ideas for improvements and further work.

10.1 Working procedure

When starting this project, the goal was to create a card playing robot. A decision was made early on to focus on creating a complete system rather than going into detail on every little part. This way of working may be referred to as the spiral model as you start by working a little bit on each component of your project, get all the parts to work together and then go into more and more detail on each part.

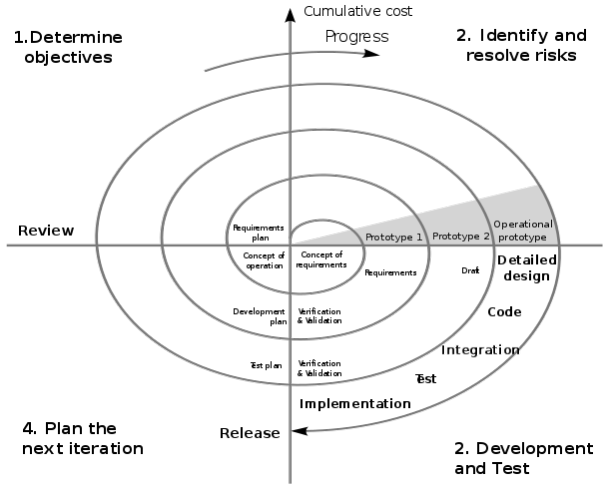


Figure 10-1: Spiral model

In hindsight it has been seen how creating a complete card playing robot in six months when having little experience working on a practical task like this, may have been a bit ambitious. One may consider if it would have been more overcoming and therefore a bit more motivating to rather pick one part of the project (like optimizing the computer vision program or setting up a robust network using XBee modules) and focus on perfecting that one part. This way of working could then have allowed for time to be more structured and follow a procedure like the waterfall model.

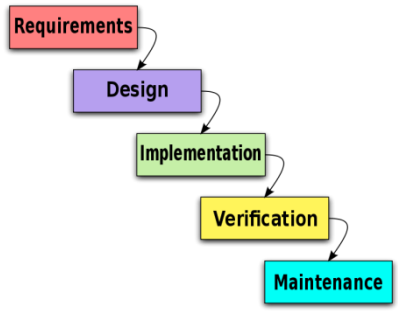


Figure 10-2: Waterfall model

Personally I prefer the spiral model. The reason is that in a project like this, everything is connected and all parts depend on each other. An example can then be made of the computer vision program. This was made more according to the waterfall model as the program was completed as a separate project before starting work on the other parts of the PokerBot. When attempting to use the program in this project, it was discovered that it required quite substantial changes for it to run fast enough on the destined HW. If the parts had been worked on simultaneously, different design choices could have been made, and the program could have been better adapted for the final HW that it is now.

Another way it would have been possible to do this project, would have been doing a purely theoretical project. This would have led to a lot less time being spent setting up the system and debugging faults. This time could then have been spent on research and therefore have led to better solutions.

10.2 Final result

Even though the final product is not quite as finished as what was hoped when starting this project, most of the presumed largest obstacles for creating a card plying robot have been overcome. The last big task still missing is the robotic arm that would be used for picking up playing cards. There had also been a hope that the poker skills of James himself would have been better than only checking whenever it is his turn.

From the response times for identifying playing cards seen in the result videos, the response times on the BeagleBoard is quite a bit slower than on the stationary computer. As the programs works now, it is assumed to be too slow for a commercialized product when running on the BeagleBoard. As discussed in chapter 4.3, this speed could be improved a lot by finding a camera better suited to the module. Also the computer vision part itself does still have great opportunities for improvement.

The way the system works now, it would only be possible to have a single James module at a poker table. This could be solved by making the James module first attempt to detect whether another James module already is working as the main module on the desired table. If this is the case, the first module should work as a player module instead of working as the main module. This means that it would not control the Texas Hold'em application or read the cards on the table. It would only pick up its pocket cards and decide on what actions to take when the game requires it.

If several different poker tables were to coexist in the same room, this would result in a problem regarding how the network is working for the current prototype. For now, the system will use the default PAN ID and channel on the XBee modules and messages that are broadcasted will be received by all players on every table in range. This should be improved by having the main module on each table, work as a coordinator when registering players. Then player could connect to the wanted coordinator and each coordinator would make sure PAN ID and channel did not collide for different tables. To achieve this, the procedure in [22] under the chapter "XBee®/XBee-PRO® Networks" can be followed.

The Texas Hold'em application seems to work quite well, but there is some missing functionality for when a player runs out of chips and is out of the game, if new players want to join an ongoing game and if players want to pause a game to continue it at a later time.

There is also a problem with the Texas Hold'em application when interpreting the output from the computer vision program. The problem is that if a card lay under the camera (say a seven of heart) and this card has been correctly detected and marked as a dead card, the card may be re detected (as maybe a seven of diamonds) if enough frames are being processed next time cards should be read. An example of this can be seen at 3:15 in the result video for the Texas Hold'em application. Here a seven of diamonds is detected as a possible pocket card. This is probably a misreading of either the jack or ace of diamond that is currently lying on the table.

Regarding the poker skills of James himself, it was planned to use the same kind of code as what is used for existing pokerbots made to play online poker. These pokerbots will evaluate every possible outcome from the current game situation and then return a percentage representing how many of these outcomes that will result in the player winning. James could then use this percentage combined with the current value of the pot, the players bankroll and how much the player has already applied to the pot to make a decision. The decision would also be affected by a random number, allowing for James to "bluff" by sometimes making decisions that does not correspond with the chance of winning.

10.2.1 How can it be used?

Picture yourself entering the activity room of a hospital you are unfortunately staying at. Scenario one is that you enter the room and one of the people in there is sitting in a chair reading a newspaper with a pile of board games, including a card deck, lying on the table next to him. What are the odds that you will engage in a conversation with this strange man reading his newspaper? Let alone what are the odds you two will sit down to play a game of poker?

Now scenario two: Once again you enter the room, but this time, the man is sitting at a table playing poker against a robot. At the table there are several chairs so that other players can join the game. Now what are the odds of you and this strange person will be playing a game of poker together, having a conversation? James the PokerBot would always be available (unless there were no more seats available at the table) and would not require any staff to operate it (but course they would be allowed to join you at your game while having a quiet night shift).

James the PokerBot could also be available at airports, arcades, schools and basically anywhere else that people sometimes needs to kill some time. It could of course also be purchased by private persons that just really like cool geeky gadgets, and let's be honest, there are many of us.

Except for the BeagleBoard, none of the components used to create James the PokerBot are too expensive. The BeagleBoard itself is a development board that can be used to create several other practical projects. As the finished system also is easily transportable, it would be possible to use this project at different events promoting cybernetics and other paths of civil engineering. It would then be used as a motivation tool targeting students that one day may choose this line of higher education, demonstrating the kind of projects they may be working on.

10.3 Ideas of improvements and further work

If there had been time for it or if work on James the PokerBot was to be continued, there are several ideas of improvements and further work.

One of the ideas that are expected to require most work is using a movable camera. For now the camera will be placed at a static position relative to the table. The camera must be close enough to be able to identify the cards in the picture while being far enough away to see a large enough part of the table. If the camera on the other hand had functionality to move (and zoom) so that if could focus on the relevant parts of the table, this would allow it to get closer to the cards (and therefore achieve more accurate identifications) while being able to see cards on the entire table. An example of how this could work would be that the camera first would scan the table from afar to detect relevant cards for then to zoom in on those cards to identify them.

Another rather big task of improvement would be enabling James to play using real life poker chips, instead of adding bets using the player modules. This would require a computer vision program used to read the poker chips (identify value as well as calculating amount of chips in a pile) as well as an arm that would pick up chips from the pot when winning, stacking them into piles in front of James and moving piles of chips to the middle of the table. This task I think would be a great project for future master students wanting to do a practical task for their thesis.

10.3.1 Player modules

So far the player modules only work for informing James on actions taken by the human players in a game of Texas Hold'em. The plan was that these modules would be used as controllers for starting and stopping the game. It would then also be possible to choose different card games, configuring options for that game (like specifying rules for games that can be played in different ways), saving an ongoing game so that it could be continued at a later time, adding and removing players from the game etc.

This would also mean that the design of the modules had to be independent of the card game that was to be played, unlike the way they now are designed only for playing Texas Hold'em. This idea is then connected to the plan that smartphones and tablets could be used as player modules. When playing using a touch screen, it would be easy to custom make the interface for each game.

In a normal game, player stats can be seen by looking at the stacks of chips each player currently possesses. As playing with James currently means excluding poker chips from the game, it would be necessary to get this information via the menu button on the player modules.

10.3.2 Communication

As mentioned, the network designed for communication between modules, should be worked on to improve stability. Another improvement that should be implemented is the use of sleep functions to lower power consumption when running the modules using a battery. More about sleep functions can be read in [22] under *Sleep Mode* on page 24.

11 Conclusion

Working on this project has over all been enormously challenging, but this in a good way. It has made me work on several different aspects of embedded solutions which have gained me a lot of valuable experience in the field.

The project has been centralized around creating a robot that would be able to play a game of cards against human players using a regular card deck. On the road to achieve this, the following tasks have been solved:

- Several algorithms have been made to create a computer vision program used to read the playing cards on the table. The program was made so that it could be configured to work with different types of card decks.
- The algorithms have been adapted to run on hardware that was assumed easily available and relatively inexpensive.
- To test that the computer vision worked in a real life card game, algorithms were created to set up a working game of Texas Hold'em.
- A network based on ZigBee was set up using (also easily available and relatively inexpensive) XBee modules. This network was made so that an undefined number of nodes (players) would be able to attend the game.
- As each player would need a way to interface with the main unit (running the Texas Hold'em and computer vision applications) over the network, simple player modules were made based on AVR controllers.

Even though the final product never got as complete as hoped when starting this project, most of the largest obstacles for creating a card playing robot have been overcome. One of the main parts still missing for the prototype is a mechanism for picking up playing cards that lie on a table. Still, the prototype will as it is now, be able to play a game of Texas Hold'em against an undefined number of human players using a regular deck of cards.

The prototype is still a long way from being a commercialized product although this is demonstrated to be feasible. However, with the addition of the mechanical arm, James would as discussed be a great asset for the department to inspire young students to choose science based studies.

12 Afterword

A card playing robot may not solve any world problems and I strongly doubt that it will result in any Nobel prizes. Still, being able to work on a project like this, has led to a great deal of personal development. I would therefore like to thank the Department of Engineering Cybernetics at NTNU, for allowing me to have this project as my master thesis the spring 2012. I would also like to thank Amund Skavhaug for having been my supervisor during the project.

13 BIBLIOGRAPHY

[1] M. Edvardsen, "James the PokerBot - "Part1: Computer Vision"," NTNU, Trondheim, 2011.

[2] M. Edvardsen, "James the PokerBot: computer vision 1.1," 2012. [Online]. Available: http://www.youtube.com/watch?v=MHM0-zkwNsg.

[3] M. Edvardsen, "James the PokerBot: version 1.0," 2012. [Online]. Available: http://youtu.be/J2RSRdu6jtU.

[4] G. Hollinger and N. Ward, "Introducing Computers to blackjack: Implementation of a Card Recognition System using Computer Vision Techniques," 2004.

[5] C. B. Zheng and R. Dr Green, "Playing Card Recognition Using Rotational Invariant Template Matching," University of Canterbury, New Zealand, 2007.

[6] CardsharkOnline, "Tangam Visual Recognition System Card Demo," 2009. [Online]. Available: http://www.youtube.com/watch?v=RgjPcP4HN58.

[7] animaltrainer88, "Playing Card Recognition Using AForge.Net," 2011. [Online]. Available: http://www.youtube.com/watch?v=dui3ftwsuhM&feature=related.

[8] purely-poker.com, "Poker Bots," 2011. [Online]. Available: http://www.purely-poker.com/pokerbot.htm. [Accessed 2012].

[9] F. T. L. M. E. L. Felix Hammer, "PokerTH," [Online]. Available: http://www.pokerth.net/.

[10] C. Simon, "CARD PLAYING ROBOT," [Online]. Available: http://www.youtube.com/watch?v=mld9swsgYFg.

[11] Maikl2811, "ABB ROBOT PLAY CARDS, BLACK JACK," [Online]. Available:

http://www.youtube.com/watch?v=JNueKvH6kDA&feature=related.

[12] Society of Robots, "ROBOT ARM PLAYING CARD DEALER," [Online]. Available: http://www.societyofrobots.com/robot_arm_card_dealer.shtml.

[13] G. Bradski and A. Kaehler, "OpenCV Wiki," [Online]. Available: http://opencv.willowgarage.com/wiki/.

[14] "gperftools," [Online]. Available: http://code.google.com/p/gperftools/.

[15] "Analyzing Application Performance by Using Profiling Tools," [Online]. Available: http://msdn.microsoft.com/en-us/library/z9z62c29.aspx.

[16] j4ck, "Angstrom-Narcissus Online Image Builder running OpenCV," [Online]. Available: http://groups.google.com/group/beagleboard/browse_thread/thread/1ec0c6585a2141c8/19e6ac4bbdd4647f?show_docid=19e6ac4bbdd4647f.

[17] T. Weaver, "Installing Angstrom on the BeagleBoard-xM," 10 2010. [Online]. Available: http://treyweaver.blogspot.com/2010/10/installing-angstrom-on-beagleboard-xm.html. [Accessed 10 02 2012].

[18] Nabax, "BeagleBoardBeginners - eLinux," [Online]. Available: http://elinux.org/BeagleBoardBeginners.

[19] technoblogical, "Samba: share Linux Folders with your windows machines," [Online]. Available: http://www.youtube.com/watch?v=p2r0kIB_ItE.

[20] T. Pitman, "Remote C / C++ development on the Beagleboard using NetBeans IDE," [Online]. Available: http://mechomaniac.com/BeagleboardDevelopmentWithNetbeans.

[21] Wikipedia, "ZigBee," [Online]. Available: http://en.wikipedia.org/wiki/ZigBee.

[22] Digi International Inc., "XBee®/XBee-PRO® RF Modules," Minnetonka, MN 55343, 2012.

[23] digi.com, "XBee Series 1 and XBee Series 2 Differences," [Online]. Available: http://www.digi.com/technology/rf-tips/2007/05.

[24] GROUND Lab Wiki, "networks_overview - GROUND Lab Wiki," 28 12 2010. [Online]. Available: http://wiki.groundlab.cc/doku.php?id=networks_overview. [Accessed 8 2 2012].

[25] PokerStars, "Texas Holdem Poker Rules," 2001. [Online]. Available: http://www.pokerstars.com/poker/games/texas-holdem/. [Accessed 2012].

[26] I. Texas Instruments, "OpenCV on TI's DSP+ARM®," [Online]. Available: http://www.ti.com/lit/wp/spry175/spry175.pdf.

[27] M-Short, "XBee Introduction and Buying Guide - SparkFun Electronics," 24 02 2011. [Online]. Available: http://www.sparkfun.com/tutorials/257. [Accessed 08 02 2012].

[28] Parallax, Inc, "WIRELESSLY NETWORKING PROPELLER CHIPS," in *PROGRAMMING AND CUSTOMIZING THE MULTICORE PROPELLER MICROCONTROLLER*, McGraw-Hill/TAB Electronics; 1 edition, 2010, pp. 189-233.

[29] j.v.d, "RS232 using thread-safe calls to Windows Forms controls," 21 1 2007. [Online]. Available: http://www.codeproject.com/Articles/17261/RS232-using-thread-safe-calls-to-Windows-Forms-con. [Accessed 20 2 2012].

[30] M. Edvardsen, "James the PokerBot: Texas Hold'em app," 2012. [Online]. Available: http://youtu.be/vYG2iN8HISE.

[31] Digi International Inc., "XBee®/XBee-PRO® RF Modules," Minnetonka, MN 55343, 2012.

14 Appendix

14.1 Poker rules

The following rules are quoted from [25].

14.1.1 Main rules

The Blinds

In Hold'em, a marker called 'the button' or 'the dealer button' indicates which player is the nominal dealer for the current game. Before the game begins, the player immediately clockwise from the button posts the "small blind", the first forced bet. The player immediately clockwise from the small blind posts the "big blind", which is typically twice the size of the small blind, but the blinds can vary depending on the stakes and betting structure being played.

In Limit games, the big blind is the same as the small bet, and the small blind is typically half the size of the big blind but may be larger depending on the stakes. For example, in a $2/$4 Limit game the small blind is $1 and the big blind is $2. In a $15/$30 Limit game, the small blind is $10 and the big blind is $15.

In Pot Limit and No Limit games, the games are referred to by the size of their blinds (for example, a $1/$2 Hold'em game has a small blind of $1 and a big blind of $2).

Depending on the exact structure of the game, each player may also be required to post an 'ante' (another type of forced bet, usually smaller than either blind, posted by all players at the table) into the pot.

Now, each player receives his or her two pocket cards. Betting action proceeds clockwise around the table, starting with the player 'under the gun' (immediately clockwise from the big blind).

Player Betting Options

In Hold'em, as with other forms of poker, the available actions are 'fold', 'check', 'bet', 'call' or 'raise'. Exactly which options are available depends on the action taken by the previous players. Each poker player always has the option to fold, to discard their cards and give up any interest in the pot. If nobody has yet made a bet, then a player may either check (decline to bet, but keep their cards) or bet. If a player has bet, then subsequent players can fold, call or raise. To call is to match the amount the previous player has bet. To raise is to not only match the previous bet, but to also increase it.

Pre-Flop
After seeing his or her pocket cards, each player now has the option to play his or her hand by calling or raising the big blind. The action begins to the left of the big blind, which is considered a 'live' bet on this round. That player has the option to fold, call or raise. For example, if the big blind was $2, it would cost $2 to call, or at least $4 to raise. Action then proceeds clockwise around the table.

Note: The betting structure varies with different variations of the game. Explanations of the betting action in Limit Hold'em, No Limit Hold'em, and Pot Limit Hold'em can be found below.

Betting continues on each betting round until all active players (who have not folded) have placed equal bets in the pot.

The Flop
Now, three cards are dealt face-up on the board. This is known as 'the flop'. In Hold'em, the three cards on the flop are community cards, available to all players still in the hand. Betting on the flop begins with the active player immediately clockwise from the button. The betting options are similar to pre-flop, however if nobody has previously bet, players may opt to check, passing the action to the next active player clockwise.

The Turn
When the betting action is completed for the flop round, the 'turn' is dealt face-up on the board. The turn is the fourth community card in Hold'em (and is sometimes also called 'Fourth Street'). Another round of betting ensues, beginning with the active player immediately clockwise from the button.

The River
When betting action is completed for the turn round, the 'river' or 'Fifth Street' is dealt face-up on the board. The river is the fifth and final community card in a Hold'em game. Betting again begins with the active player immediately clockwise from the button, and the same betting rules apply as they do for the flop and turn, as explained above.

The Showdown
If there is more than one remaining player when the final betting round is complete, the last person to bet or raise shows their cards, unless there was no bet on the final round in which case the player immediately clockwise from the button shows their cards first. The player with the best five-card poker hand wins the pot. In the event of identical hands, the pot will be equally divided between the players with the best hands. Hold'em rules state that all suits are equal.

After the pot is awarded, a new hand of Hold'em is ready to be played. The button now moves clockwise to the next player, blinds and antes are once again posted, and new hands are dealt to each player.

14.1.2Limit, No Limit, Pot Limit and Mixed Texas Hold'em

Hold'em rules remain the same for Limit, No Limit and Pot Limit poker games, with a few exceptions:

Limit Texas Hold'em

Betting in Limit Hold'em is in pre-determined, structured amounts. Pre-flop and on the flop, all bets and raises are of the same amount as the big blind. On the turn and the river, the size of all bets and raises doubles. In Limit Hold'em, up to four bets are allowed per player during each betting round. This includes a (1) bet, (2) raise, (3) re-raise, and (4) cap (final raise).

No Limit Texas Hold'em

The minimum bet in No Limit Hold'em is the same as the size of the big blind, but players can always bet as much more as they want, up to all of their chips.

Minimum raise: In No Limit Hold'em, the raise amount must be at least as much as the previous bet or raise in the same round. As an example, if the first player to act bets $5 then the second player must raise a minimum of $5 (total bet of $10).

Maximum raise: The size of your stack (your chips on the table).

In No Limit Hold'em, there is no 'cap' on the number of raises allowed.

Pot Limit Texas Hold'em

The minimum bet in Pot Limit Hold'em is the same as the size of the big blind, but players can always bet up to the size of the pot.

Minimum raise: The raise amount must be at least as much as the previous bet or raise in the same round. As an example, if the first player to act bets $5 then the second player must raise a minimum of $5 (total bet of $10).

Maximum raise: The size of the pot, which is defined as the total of the active pot plus all bets on the table plus the amount the active player must first call before raising.

Example: If the size of the pot is $100, and there is no previous action on a particular betting round, a player may bet a maximum of $100. After that bet, the action moves to the next player clockwise. That player can either fold, call $100, or raise any amount between the minimum ($100 more) and the maximum. The maximum bet in this case is $400 - the raiser would first call $100, bringing the pot size to $300, and then raise $300 more, making a total bet of $400.

In Pot Limit Hold'em, there is no 'cap' on the number of raises allowed.

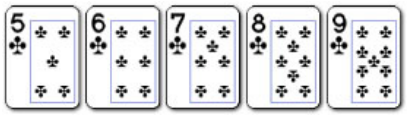
Mixed Texas Hold'em

In Mixed Hold'em, the game switches between rounds of Limit Hold'em and No Limit Hold'em. The blinds are typically increased when the game switches from No Limit to Limit, to ensure some consistency in the average pot size in each game. The betting rules on each round follow the rules for that game, as described above.

14.1.3Hand ranking

Traditional High Poker Hand Ranks

Straight Flush: Five cards in sequence, of the same suit.



In the event of a tie: Highest rank at the top of the sequence wins.

The best possible straight flush is known as a **royal flush**, which consists of the ace, king, queen, jack and ten of a suit. A royal flush is an unbeatable hand.

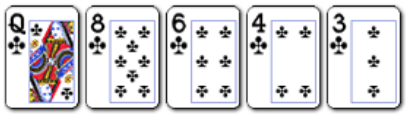
Four of a Kind: Four cards of the same rank, and one side card or 'kicker'.

In the event of a tie: Highest four of a kind wins. In community card games where players have the same four of a kind, the highest fifth side card ('kicker') wins.

Full House: Three cards of the same rank, and two cards of a different, matching rank.

In the event of a tie: Highest three matching cards wins the pot. In community card games where players have the same three matching cards, the highest value of the two matching cards wins.

Flush: Five cards of the same suit.



In the event of a tie: The player holding the highest ranked card wins. If necessary, the second-highest, third-highest, fourth-highest, and fifth-highest cards can be used to break the tie. If all five cards are the same ranks, the pot is split. The suit itself is never used to break a tie in poker.

Straight: Five cards in sequence.

In the event of a tie: Highest ranking card at the top of the sequence wins.

Note: The Ace may be used at the top or bottom of the sequence, and is the only card which can act in this manner. A,K,Q,J,T is the highest (Ace high) straight; 5,4,3,2,A is the lowest (Five high) straight.

Three of a kind: Three cards of the same rank, and two unrelated side cards.

In the event of a tie: Highest ranking three of a kind wins. In community card games where players have the same three of a kind, the highest side card, and if necessary, the second-highest side card wins.

Two pair: Two cards of a matching rank, another two cards of a different matching rank, and one side card.

In the event of a tie: Highest pair wins. If players have the same highest pair, highest second pair wins. If both players have two identical pairs, highest side card wins.

One pair: Two cards of a matching rank, and three unrelated side cards.

In the event of a tie: Highest pair wins. If players have the same pair, the highest side card wins, and if necessary, the second-highest and third-highest side card can be used to break the tie.

High card: Any hand that does not qualify under a category listed above.

In the event of a tie: Highest card wins, and if necessary, the second-highest, third-highest, fourth-highest and smallest card can be used to break the tie.

14.2 Code referenced to in the report

14.2.1 Card recognition

14.2.1.1 Improved rank location

```
segmentSize = 0;
tempSegmentSize = 0;
tempMin = valueMin.y;
tempMax = valueMin.y;
lastBreak = valueMin.y;

for( y = valueMin.y; y < valueMax.y; y++ ) {
      separationLineValue = 0;
      for( x = valueMin.x; x < valueMax.x; x++ )
            separationLineValue += WHITE - objIn[x + y * 50];

      if( separationLineValue < (WHITE * 2) ) { //if less than two
black pixels
            if( tempSegmentSize > segmentSize ) {
                  tempMin = lastBreak;
                  tempMax = y;
                  segmentSize = tempSegmentSize;
                  tempSegmentSize = 0;
            }
            lastBreak = y + 1;
      }
      tempSegmentSize++;
}
if( tempSegmentSize > segmentSize ) {
      tempMin = lastBreak;
      tempMax = y;
}

valueMin.y = tempMin;
valueMax.y = tempMax;
```

14.2.1.2 sameCardAs

```
bool Card::sameCardAs( Card *card2 )
{
      Corner corner1, corner2;
      cornerRelation cornersOnSameCard;

      corner1 = getFirstCorner( );
      corner2 = card2->getFirstCorner( );

      // See if the corner is in a position that matches what is
expected
      // for the card.
      cornersOnSameCard = cornerPositionMatchesCard(corner2);
      if(cornersOnSameCard == NOT_RELATED)
            return false;

      if( cornersOnSameCard == SAME )
            if (corner1.usesSameLines(corner2))
```

```
                return true;
        else if( cornersOnSameCard == OPOSITE )
                if (corner1.hasSameLine(corner2) &&
corner1.hasParallellLine(corner2))
                return true;
        else if( cornersOnSameCard == DIAGONAL )
                if (corner1.hasTwoOrthogonalLines(corner2))
                return true;
        return false;
}
```

14.2.2 Player module simulator - SerialDataReceivedEvent

```
Void serialPort1_DataReceived(System::Object^  sender,
SerialDataReceivedEventArgs^  e)
{
        String^ comMsg = serialPort1->ReadLine();
        String^ text = "";
        int temp;

        switch(comMsg[0]){
        case 'T':  //Talking
            Invoke(gcnew EventHandler(this, &Form1::SetActive), "");
            Invoke(gcnew EventHandler(this, &Form1::EnableButtons),
"");
            break;
        case 'B':  //Bid round complete
            pot = int::Parse(comMsg->Substring(1));
            bet = 0;
            curBet = 0;
            break;
        case 'A':  //Activate
            Invoke(gcnew EventHandler(this, &Form1::SetActive), "");
            Invoke(gcnew EventHandler(this, &Form1::DisableButtons),
"");
            break;
        case 'N':  //New round
            Invoke(gcnew EventHandler(this, &Form1::SetUnActive),
"");
            Invoke(gcnew EventHandler(this, &Form1::DisableButtons),
"");
            bigBlind= int::Parse(comMsg->Substring(2, comMsg-
>IndexOf(" ")-1));
            bankRoll = int::Parse(comMsg->Substring(comMsg-
>IndexOf(" ")+1));
            switch(comMsg[1]){
            case 'N':  // Normal
                  text = "1New round";
                  state = 0;
                  bet = 0;
                  break;
            case 'D':  // Dealer
                  text = "2Task: Dealer";
                  state = 1;
```

```
                bet = 0;
                break;
          case 'S':  // Small blind
                text = "2Task: SmallBlind";
                state = 2;
                bet = bigBlind/2; //NB - not correct...
                break;
          case 'B':  // Big blind
                text = "2Task: BigBlind";
                state = 3;
                bet = bigBlind;
                break;
          default:
                text = "2Unkown msg!";
                state = -1;
          }
          highBet = bigBlind;
          curBet = bet;
          pot = 0;
          break;
    case 'U':  //Update
          switch(comMsg[1]){
          case 'H':  // Highbet
                highBet = int::Parse(comMsg->Substring(2));
                break;
          case 'B':  // Bankroll
                bankRoll = int::Parse(comMsg->Substring(2));
                break;
          }
          break;
    case 'D': // Debug
          switch(comMsg[1]){
          case 'B': // Bankroll
                temp = int::Parse(comMsg->Substring(2));
                if(temp != bankRoll)
                      text = "Bank not match";
                break;
          default:
                break;
          }
          break;
    case 'I':  //Info as text
          if(comMsg[1] == 'L'){
                switch(comMsg[2]){
                case '1': // Line 1
                      text = "1" + comMsg->Substring(3);
                      break;
                case '2': // Line 2
                      text = "2" + comMsg->Substring(3);
                      break;
                default:
                      text = "1" + comMsg->Substring(2);
                      break;
                }
```

```
            }
            else
                  text = "1" + comMsg->Substring(1);
            break;
      default:
            state = -1;
      }

      Invoke(gcnew EventHandler(this, &Form1::UpdateTextbox),
text);
 }
```

14.2.3 Player modules

14.2.3.1 LCD_putstr – prototype 1

```
#define CURSOR_HOME           0x02  // Set DDRAM address to 0
#define CUR_DISP_SHIFT_SL     0x18  // Shift display. Shift left
#define LINE1                 0x80  // Sets the DDRAM address =
0x00
#define LINE2                 0xC0  // Sets the DDRAM address =
0x40


/*--------------- Function show string message ---------------*/
void LCD_putstr( char line, char *p)
{
      LCD_clear(line);

      LCD_command(CURSOR_HOME);
      LCD_command(line); // Set address to start of line

      int cursor = 0;

      while(*p){
            LCD_text(*p++);  // Send data to LCD
            if (cursor > 15){ // If data exceeds 16 characters,
shift display left
                  _delay_ms(60);
                  LCD_command(CUR_DISP_SHIFT_SL);
            }
            cursor++;
      }
      _delay_ms(10);
}
```

14.2.3.2 handleSw – prototype 1

```
void handleSw()
{
      int minRaise;
      char data[10];

      minRaise = thePlayer.highBet + thePlayer.bigBlind;
```

```
	switch ( PINA ){
	case FOLD:
		if(thePlayer.active){
			xBee_Transmit( 'F' );
			xBee_Transmit( 'X' );
			thePlayer.active = 0;
			thePlayer.bet = thePlayer.curBet;
			player_updateBet();
		}
		break;
	case CALL:
		if(thePlayer.active){
			xBee_Transmit( 'C' );
			xBee_Transmit( 'X' );
			thePlayer.bet = thePlayer.highBet;
			thePlayer.curBet = thePlayer.bet;
			thePlayer.active = 0;
			player_updateBet();
		}
		break;
	case RAISE:
		if(thePlayer.active){
			data[0] = 'R';
			if(thePlayer.bet < minRaise)
				thePlayer.bet = minRaise;
			itoa(thePlayer.bet, (data + 1), 10);
			xBee_WriteLine( data );
			thePlayer.curBet = thePlayer.bet;
			thePlayer.active = 0;
			player_updateBet();
		}
		break;
	case UP:
		handleUpSw();
		break;
	case DOWN:
		handleDownSw();
		break;
	case MENU:
		xBee_Transmit( 'M' );
		xBee_Transmit( 'X' );
		break;
	case SW6:
		LCD_clear(LINE1);
		LCD_clear(LINE2);
		break;
	case SW7:
		player_updateText();
		break;
	default:
		break;
	}
}
```

14.2.3.3 getButton – prototype 2

```
#define FOLD  0x01
#define CALL  0x02
#define RAISE 0x04
#define MENU  0x20

#define SW_MENU GPIO_PUSH_BUTTON_0
#define SW_RAISE GPIO_PUSH_BUTTON_1
#define SW_CALL GPIO_PUSH_BUTTON_2

uint8_t getButton()
{
      if (!ioport_get_value(SW_RAISE))
            return RAISE;
      else if (!ioport_get_value(SW_CALL))
            return CALL;
      else if (!ioport_get_value(SW_MENU))
            return MENU;
      else if (check_touch_key_pressed( ))
            return FOLD;
      else
            return 0x00;
}
```

14.2.4 Communication

14.2.4.1 Player registering – Player modules

```
bool registerAsPlayer()
{
      LCD_putstr(LINE1, "Registering");
      if(!detectMainModule())
            return false;

      LCD_putstr(LINE1, "Detected James");
      if(!connectToMainModule())
            return false;

      LCD_putstr(LINE3, "Registered");
      return true;
}

bool detectMainModule()
{
      int i;
      bool foundCoordinator;
      Byte addressH[9], addressL[9];

      for (i = 0; i < 50; i++){
            foundCoordinator = xBee_getCoordinator(addressH,
addressL);
            if(foundCoordinator){
                  xBee_setDestinationAddr(addressH, addressL);
                  return true;
```

```
            }
            _delay_ms(100);
      }

      return false;
}

bool connectToMainModule()
{
      int i;
      bool connected;

      for (i = 0; i < 5; i++){
            xBee_sendMySourceAddr();
            LCD_putstr(LINE2, "Sent address");
            xBee_waitForData(4000);
            LCD_putstr(LINE2, "Getting ACK");
            connected = xBee_getConnectedConfirm();
            if(connected)
                  return true;
      }

      xBee_resetDestAddr();
      return false;
}
```

14.2.4.2 Player registering – Main module

```
void TexasHoldem::getPlayers( )
{
      GetPlayerStat stat;
      char addressL[9], addressH[9];

      james = new James( 5000 );
      players.push_back( james );
      nrOfPlayers = 1;

      comDevice->broadcast(SOURCE_B);
      while(1){
            stat = comDevice->getPlayer(addressH, addressL);
            if(stat == IS_PLAYER)
                  addPlayer(addressH, addressL);
            else if(stat == IS_READYTOPLAY)
                  break;
            else
                  comDevice->broadcast(SOURCE_B);
      }
      comDevice->clearMsgList( PLAYER_T );
}
```

14.2.5 readCards

```
void readCards(BoardCard oldCards[], unsigned short nrOfCards,
Board *theBoard)
{
      vector<BoardCard> newCards;
```

```
	vector<BoardCard>::iterator newCard;

	int i;
	string result;
	bool updated[nrOfCards];
	float fact;

	// Initialize that none of the previously
	// found cards has been updated by the
	// new result string..
	fillArray(updated, false, nrOfCards);

	// Find playing cards in frame
	playingCardRecognizing(result);

	// Remove cards already on the table
	theBoard->removeDeadCards(result);

	// If no cards are found, decrease strengt
	// of previously updated cards
	if( result.length() == 0 ){
		for( i = 0; i < nrOfCards; i++ ){
			if( oldCards[i].valueStrength > 0 )
				oldCards[i].valueStrength -= 10;
			if( oldCards[i].suitStrength > 0 )
				oldCards[i].suitStrength -= 10;
		}
		return;
	}

	// NB - do this after removing re updated cards?
	// If more cards are found than needed
	// Get the amount of cards searched for as the
	// "nrOfCards" strongest remaining cards in the srting
	if( result.length() > (unsigned)(LENGTH_OF_CARD * nrOfCards)
)
		getStrongestCardsFromString( result, nrOfCards );

	for(i = 0; i < (int)result.length()/14; i++){
		newCards.push_back(BoardCard());
		newCards[i].getFromString(result, i);
	}

	//decrese cards not found..
	for( i = 0; i < nrOfCards; i++ ){
		newCard = newCards.begin();
		while ( newCard++ != newCards.end() )
			if( oldCards[i].isCloseTo(*newCard) )
				break;
		if( newCard == newCards.end() ){ //found no cards close
to previous cards
			if( oldCards[i].valueStrength > 0 )
				oldCards[i].valueStrength -= 10;
			if( oldCards[i].suitStrength > 0 )
```

```
                        oldCards[i].suitStrength -= 10;
                }
        }

        // Increse value of cards previously detected
        for( i = 0; i < nrOfCards; i++ ){
                newCard = newCards.begin();
                while ( newCard != newCards.end() ){
                        if( oldCards[i].hasSameValueAndSuitAs(*newCard)
){
                                fact = 1;
                                if(!oldCards[i].isCloseTo(*newCard))
                                        fact = 0.5f;
                                if( oldCards[i].valueStrength < 180 )
                                        oldCards[i].valueStrength
                                                += newCard->valueStrength*fact;
                                if( oldCards[i].suitStrength < 300 )
                                        oldCards[i].suitStrength
                                                += newCard->suitStrength*fact;
                                oldCards[i].centerPoint.x =
                                        (oldCards[i].centerPoint.x
                                                + newCard->centerPoint.x)/2;
                                oldCards[i].centerPoint.y =
                                        (oldCards[i].centerPoint.y
                                                + newCard->centerPoint.y)/2;
                                updated[i] = true;
                                newCard = newCards.erase(newCard);
                        }
                        else
                                newCard++;
                }
        }

        if(newCards.empty())
                return;

        // Update value of cards detected in the same plase as old
cards
        for(i = 0; i < nrOfCards; i++){
                newCard = newCards.begin();
                while ( newCard != newCards.end() )
                        if( oldCards[i].isCloseTo(*newCard) &&
!updated[i] ){
                                if( oldCards[i].hasSameValueAs(*newCard) ){
                                        if( oldCards[i].valueStrength < 180 )
                                                oldCards[i].valueStrength
                                                        += newCard-
>valueStrength;
                                }
                                else{
                                        if( oldCards[i].valueStrength > 0 )
                                                oldCards[i].valueStrength
                                                        -= newCard-
>valueStrength;
```

```
                              if( oldCards[i].valueStrength
                                    <= newCard->valueStrength)
                                    oldCards[i].value = newCard-
>value;
                        }

                        if( oldCards[i].hasSameSuitAs(*newCard)){
                              if( oldCards[i].suitStrength < 300 )
                                    oldCards[i].suitStrength
                                          += newCard->suitStrength;
                        }
                        else{
                              if( oldCards[i].suitStrength > 0 )
                                    oldCards[i].suitStrength
                                          -= newCard->suitStrength;
                              if( oldCards[i].suitStrength
                                    <= newCard->suitStrength)
                                    oldCards[i].suit = newCard-
>suit;
                        }
                        updated[i] = true;
                        newCard = newCards.erase(newCard);
                  }
                  else
                        newCard++;
      }

      if(newCards.empty())
            return;

      // Add new cards
      while( !newCards.empty() ){
            newCard = newCards.begin();
            i = getWeakestCardNotUpdated( oldCards, nrOfCards,
updated );
            if( i != -1 ){
                  if( oldCards[i].valueStrength > 0 )
                        oldCards[i].valueStrength -= newCard-
>valueStrength;
                  if( oldCards[i].suitStrength > 0 )
                        oldCards[i].suitStrength -= newCard-
>suitStrength;


      if((oldCards[i].valueStrength+oldCards[i].suitStrength) <
                              (newCard->valueStrength+newCard-
>suitStrength))
                        oldCards[i] = *newCard;
                  updated[i] = true;
            }
            newCard = newCards.erase(newCard);
      }
}
```

14.2.6 handValue

```
// CONSTANTS:
#define STRAIGHT_FLUSH  920000
#define FOUR_OF_A_KIND  910000
#define FULL_HOUSE      900000
#define FLUSH           460000
#define STRAIGHT        450000
#define THREE_OF_A_KIND 440000
#define TWO_PAIR        430000
#define ONE_PAIR        400000
#define HIGH_CARD       0

// POWERS of 13
#define TT1 13
#define TT2 169
#define TT3 2197
#define TT4 28561
#define TT5 371293

int handValue( BoardCard cards[] )
{
int suit[7], value[7], flushCards[7];
int isFlush, isStraightFlush, isStraight;
int flushSuit = -1, straightHighCard = -1, straightFlushHighCard =
-1;
int quadruple = -1, triple = -1, pair[2] = {-1, -1};
int i, j;
int kick1, kick2, kick3;
int club = 0, diamond = 0, heart = 0, spade = 0;
int tmp1, tmp2;
int straightHistogram[13], straightFlushHistogram[13];
int straightFlushCount = 0, straightCount = 0;

//set values[] & suits[]
for( i = 0; i < 7; i++ ) {
      value[i] = cards[i].value - 2;
      if( value[i] < 0 )
            value[i] = 12;
}

for( i = 0; i < 7; i++ )
      suit[i] = cards[i].suitAsNr( );

// sort by increasing values
for( i = 0; i < 6; i++ )
      for( j = 0; j < 6 - i; j++ )
            if( value[j + 1] < value[j] ) {
                  tmp1 = value[j];
                  tmp2 = suit[j];
                  value[j] = value[j + 1];
                  value[j + 1] = tmp1;
                  suit[j] = suit[j + 1];
                  suit[j + 1] = tmp2;
```

```
            }

//get suit counts
for( i = 0, j = 0; i < 7; i++ ) {
      if( suit[i] == 0 )
            club++;
      else if( suit[i] == 1 )
            diamond++;
      else if( suit[i] == 2 )
            heart++;
      else
            spade++;
}

//check for a flush
isFlush = 1;
if( club > 4 )
      flushSuit = 0;
else if( diamond > 4 )
      flushSuit = 1;
else if( heart > 4 )
      flushSuit = 2;
else if( spade > 4 )
      flushSuit = 3;
else
      isFlush = 0;

//check for a straight flush:
if( isFlush ) {
      for( i = 0; i < 7; i++ )
            flushCards[i] = -1;

      for( i = 0, j = 6; j >= 0; j-- )
            if( suit[j] == flushSuit )
                  flushCards[i++] = value[j];

      for( i = 0; i < 13; i++ )
            straightFlushHistogram[i] = 0;

      for( i = 0; i < 7 && flushCards[i] != -1; i++ )
            straightFlushHistogram[flushCards[i]]++;

      isStraightFlush = 0;
      for( i = 0; i < 13; i++ ) {
            if( straightFlushHistogram[i] ) {
                  straightFlushCount++;
                  if( straightFlushCount >= 5 ) {
                        isStraightFlush = 1;
                        straightFlushHighCard = i;
                  }
            }
            else
                  straightFlushCount = 0;
}
```

```
}

// straight flush:
if( isStraightFlush )
      return (STRAIGHT_FLUSH + straightFlushHighCard);

//check for a straight or 4 of a kind:
for( i = 0; i < 13; i++ )
      straightHistogram[i] = 0;

for( i = 0; i < 7; i++ ) {
      straightHistogram[value[i]]++;
      if( straightHistogram[value[i]] == 4 )
            quadruple = value[i];
}

isStraight = 0;
for( i = 0; i < 13; i++ ) {
      if( straightHistogram[i] ) {
            straightCount++;
            if( straightCount >= 5 ) {
                  isStraight = 1;
                  straightHighCard = i;
            }
      }
      else
            straightCount = 0;
}

// four of a kind:
if( quadruple != -1 ) {
      i = 7;
      while ( value[--i] == quadruple ) {}
      kick1 = value[i];
      return (FOUR_OF_A_KIND + TT1*triple + kick1);
}

//check for trips and pairs:
for( i = 12; i >= 0; i-- ) {
      if( straightHistogram[i] == 3 ) {
            if( triple == -1 )
                  triple = i;
            else if( pair[0] == -1 )
                  pair[0] = i;
      }
      else if( straightHistogram[i] == 2 ) {
            if( pair[0] == -1 )
                  pair[0] = i;
            else if( pair[1] == -1 )
                  pair[1] = i;
      }
}

// full house:
```

```
if( triple != -1 && pair[0] != -1 )
      return (FULL_HOUSE + TT1*triple + pair[0]);

// flush:
if( isFlush )
      return (FLUSH + TT4*flushCards[0] + TT3*flushCards[1] +
TT2*flushCards[2]
      + TT1*flushCards[3] + flushCards[4]);

// straight:
if( isStraight )
      return (STRAIGHT + straightHighCard);

// three of a kind:
if( triple != -1 ) {
      i = 6;
      while ( value[i] == triple )
            i--;
      kick1 = value[i--];
      while ( value[i] == triple )
            i--;
      kick2 = value[i];
      return (THREE_OF_A_KIND + TT2*triple + TT1*kick1 + kick2);
}

// two pair:
if( pair[1] != -1 ) {
      i = 6;
      while ( value[i] == pair[0] || value[i] == pair[1] )
            i--;
      kick1 = value[i];
      return (TWO_PAIR + TT2*pair[0] + TT1*pair[1] + kick1);
}

// one pair:
if( pair[0] != -1 ) {
      i = 6;
      while ( value[i] == pair[0] )
            i--;
      kick1 = value[i--];
      while ( value[i] == pair[0] )
            i--;
      kick2 = value[i--];
      while ( value[i] == pair[0] )
            i--;
      kick3 = value[i];
      return (ONE_PAIR + TT3*pair[0] + TT2*kick1 + TT1*kick2 +
kick3);
}

// high card (no pair):
return (HIGH_CARD + TT4*value[6] + TT3*value[5] + TT2*value[4]
            + TT1*value[3] + value[2]);
}
```

14.2.7 Windows debug app

The following code was executed every time an event occurred on the serial port connected to the app.

```
wchar_t card[2];
String^ comMsg = serialPort1->ReadLine();

switch(comMsg[0]){
case 'H': // Hand card
      switch(comMsg[1]){
      case '1': // Card one
            card[0] = comMsg[2];
            card[1] = comMsg[3];
            pictureBox6->Image = imageList1-
>Images[stringToCard(card)];
            break;
      case '2': // Card two
            card[0] = comMsg[2];
            card[1] = comMsg[3];
            pictureBox7->Image = imageList1-
>Images[stringToCard(card)];
            break;
      default:
            Invoke(gcnew EventHandler(this,
&Form1::updateLabelText),
                  "Unknown hand card");
      }
      break;
case 'F': // Flop card
      card[0] = comMsg[2];
      card[1] = comMsg[3];
      switch(comMsg[1]){
      case '1': // Card one
            pictureBox1->Image = imageList1-
>Images[stringToCard(card)];
            break;
      case '2': // Card two
            pictureBox2->Image = imageList1-
>Images[stringToCard(card)];
            break;
      case '3': // Card three
            pictureBox3->Image = imageList1-
>Images[stringToCard(card)];
            break;
      default:
            Invoke(gcnew EventHandler(this,
&Form1::updateLabelText),
                  "Unknown flop card");
      }
break;
case 'T': // Turn
      card[0] = comMsg[1];
      card[1] = comMsg[2];
```

```
      pictureBox4->Image = imageList1->Images[stringToCard(card)];
      break;
case ‘R’: // River
      card[0] = comMsg[1];
      card[1] = comMsg[2];
      pictureBox5->Image = imageList1->Images[stringToCard(card)];
      break;
case ‘I’: // Info
      switch(comMsg[1]){
      case ‘1’:
            Invoke(gcnew EventHandler(this,
&Form1::updateLabelText),
                  comMsg->Substring(2));
            break;
      case ‘2’:
            break;
            default:
            break;
      }
      break;
case ‘C’: // Clear
      pictureBox1->Image = nullptr;
      pictureBox2->Image = nullptr;
      pictureBox3->Image = nullptr;
      pictureBox4->Image = nullptr;
      pictureBox5->Image = nullptr;
      pictureBox6->Image = nullptr;
      pictureBox7->Image = nullptr;
      break;
default:
      Invoke(gcnew EventHandler(this, &Form1::updateLabelText),
“Unknown input”);
 }
```