Lego Mindstorms NXT & CMUcam3

This document describes the setup and some technical details for using the CMUcam3 [1] [2] [3] as a vision sensor for the Lego Mindstorms NXT. The essential module for coupling the CMUcam3 with the Lego Mindstorms NXT is an I^2C-to-RS232 bridge that is used to establish the communication between the two components. The document has been written with two goals in mind. First, it summarizes the essential steps how to set up a Lego Mindstorms robot with the CMUcam3 as a vision sensor using the mentioned I^2C-to-RS232 bridge. Second, it describes the underlying function, such that the reader should be able to understand how the system works and how to modify it for other purposes than those described in this document.
In the first section, the software environment and CMUcam3 setup is described. In the second section, it is described how the CMUcam3 can be used from a standard PC using the CMUcam2 GUI. In the third section, it is described how the CMUcam3 can be used with the Lego Mindstorms NXT and some details on the I^2C-to-RS232 bridge are given that is used for connecting the Lego Mindstorms NXT with the CMUcam3.
The work described in this document is an extension of the work described in the semester thesis of Leo den Hartog [7]. The advantage is a more versatile communication protocol, resulting in easier use of the system (the standard firmware of the CMUcam3 can be used, for instance) and higher update rates.

Section 1

Software Environment and CMUcam3 Setup

This section describes the software environment and CMUcam3 setup required in case the I^2C-to-RS232 is already available. Tools required for the development and debugging of the I^2C-to-RS232 bridge are not described. Further information about the CMUcam3 setup is available in a separate document [4].

1.1 Software Environment Setup

The software environment described in the following paragraphs has been tested under Windows XP. Corresponding versions of the tools should also be available for Linux and Mac platforms.

1.1.1 LPC2000 Flash Utility

The LPC2000 Flash Utility is needed to flash the LPC2106B ARM processor on the CMUcam3 with firmware. Download the LPC2000 Flash Utility provided by NXP from the following location and install it:
http://www.nxp.com/files/markets/microcontrollers/philips_flash_utility.zip

1.1.2 CMUcam2 GUI

The CMUcam2 GUI is a graphic frontend for controlling the CMUcam3 from a standard PC. Currently, there is no special GUI for the CMUcam3. By using the appropriate firmware on the CMUcam3, it can emulate the CMUcam2 and can thus be controlled by the CMUcam2 GUI. The CMUcam2 GUI can be downloaded from the following location:
http://www.cs.cmu.edu/~cmucam2/downloads.html
The CMUcam2 GUI is Java-based and does not require installation.

1.1.3 Brick Command Center

The Brick Command Center is an integrated development environment for developing NXC (not exactly C) programs for the Lego Mindstorms NXT. Download the Brick Command Center from the following location and install it:
http://bricxcc.sourceforge.net

1.2 CMUcam3 Setup

To use the CMUcam3, connect it to a DC power supply of 6 to 15 volts that is capable of supplying at least 150 milliamperes of current (such as a power adapter or a 9 volt battery), see Figure 1. (Internally, this power supply is filtered by a 5 volt regulator.)

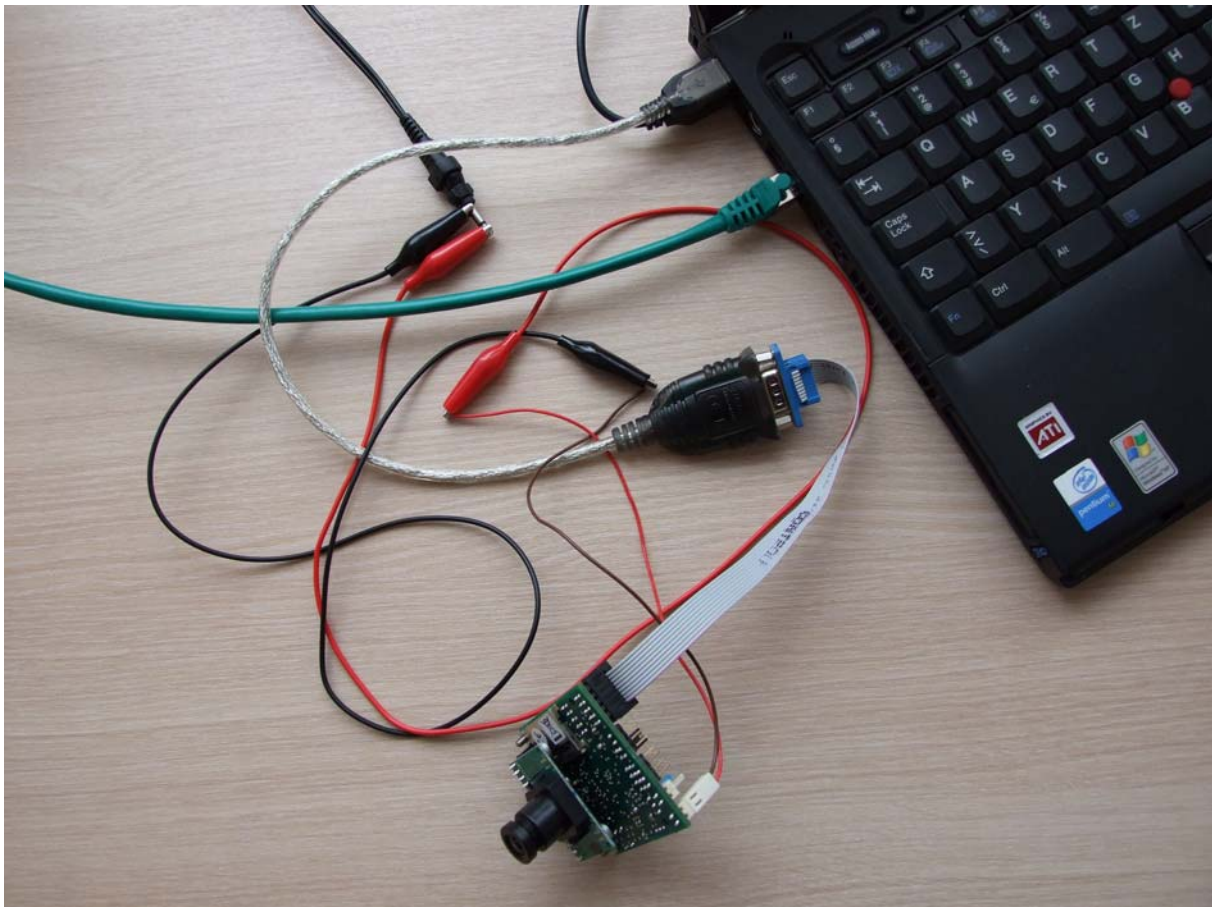

Figure 1: CMUCam3 connected to a PC via a USB-to-serial cable. The colors of the power supply cables are the "usual" ones: black for GND and red for +6V.

For data I/O, the RS232 port of the camera is used. Connect the RS232 connector of the CMUcam3 directly with the serial port of your computer in case it has got one. Unfortunately, today's laptops usually do not have serial ports any more. To connect the CMUcam3 to the PC, one can use a USB-to-serial cable, as shown in Figure 1. We used a cable based on the PL-2303 chip offered by Prolific Technology. A selection of cables where this chip is used is available at: http://www.prolific.com.tw/eng/downloads.asp?ID=31
After installing the driver that comes with the cable, an according COM (serial) port shows up in the device manager, as shown in Figure 2.

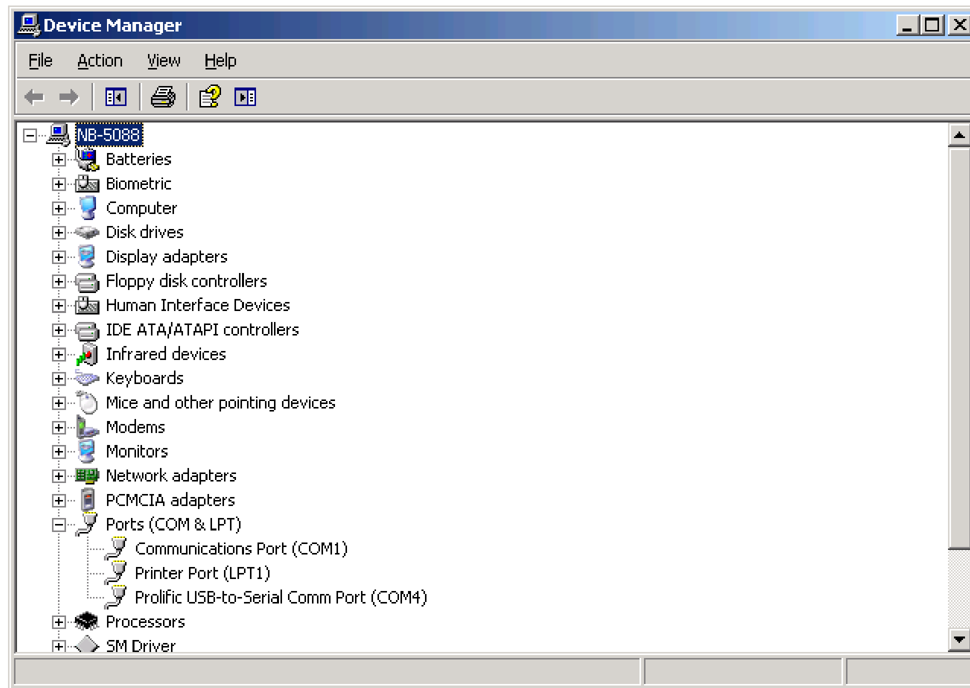

Figure 2: Microsoft Windows XP device manager showing correct installation of USB-to-serial cable.

Section 2

Using the CMUcam3 in CMUcam2 Emulation Mode

The easiest way to use the CMUcam3 is by using it in CMUcam2 emulation mode. For doing this, first the CMUcam2 emulation firmware needs to be downloaded to the camera. Afterwards, the CMUcam2 GUI (a Java-based graphical tool) can be used to take pictures with the camera and try different features, such as tracking a colored object.

To download the CMUcam2 emulation firmware, proceed as follows:

- Download the hex image of the CMUcam2 emulation firmware from: http://www.cmucam.org/wiki/cmucam2-emulation
  For operation in connection with the NXT, the firmware version with a baudrate of 19200 bit/s is needed (this is a constraint of the I^2C-to-RS232 bridge).
- Start the LPC2000 Flash Utility.
- Turn on the CMUcam3 while pressing the reset button to startup in programming mode.
- Upload the hex image to the CMUcam3 using the “Upload to Flash” button. Upon successful completion, the Flash Utility window should look similar to the one shown in Figure 3.
- Turn the CMUcam3 off and on again (without pressing the reset button) to use it. The power LED should shine brightly, while two other LEDs (of totally four) should shine dimly.

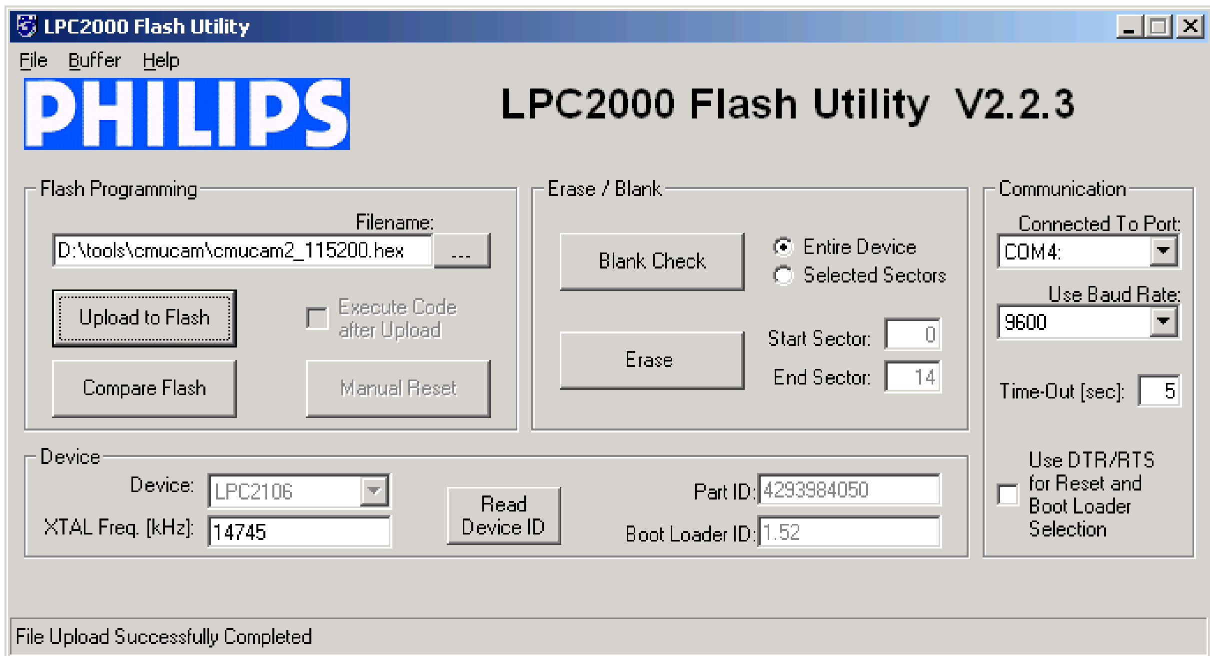

Figure 3: Philips Flash Utility after successful upload of the hex image onto the CMUcam.

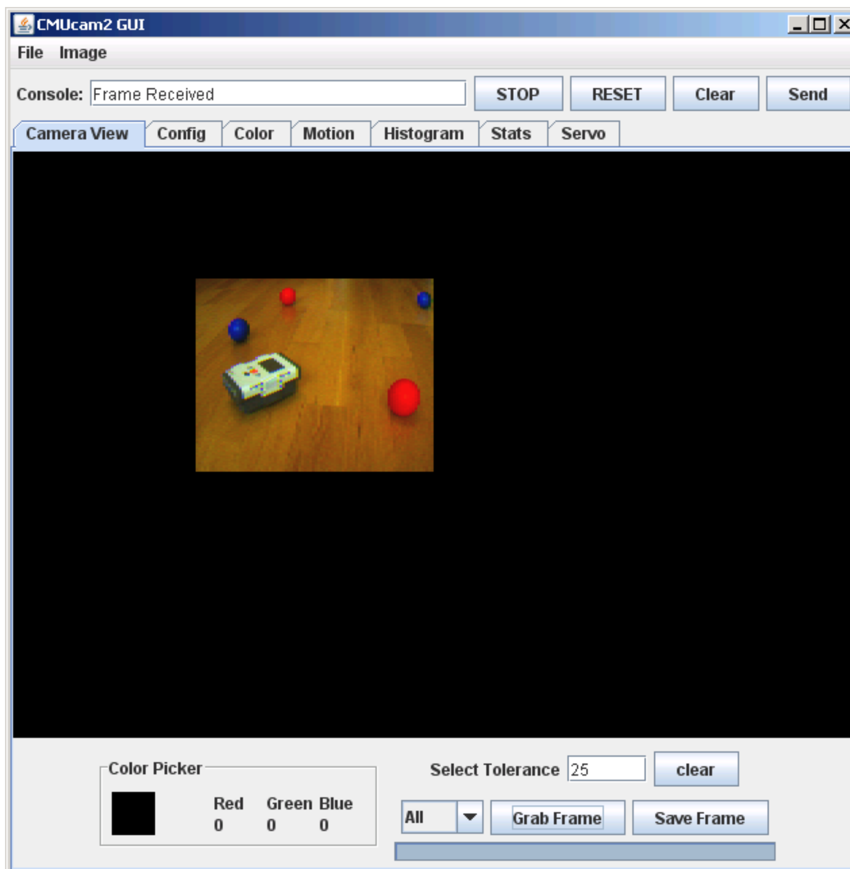

Figure 4: CMUcam2 GUI after acquiring a low resolution image from the CMUcam3.

To start the CMUcam2 GUI simply double-click on CMUcam2GUI.jar or start it from a commandline using java –jar CMUcam2GUI.jar. After entering the COM port to which the CMUcam3 is connected, the CMUcam2 GUI can be used to take pictures and try out different camera features, such as tracking a colored object, see Figure 4, Figure 5, and Figure 6. For

further information about the CMUcam2 and the CMUcam2 GUI the reader is referred to the according documents [5] [6].
Note that the CMUcam2 GUI assumes a RS232 baudrate of 115200 bits/s. To use it with a camera whose baudrate is 19200 bits/s, the CMUcam2 GUI needs to be modified and recompiled. For doing this, simply replace the parameter "5" by "2" in line 50 of SerialComm.java and recompile the package.

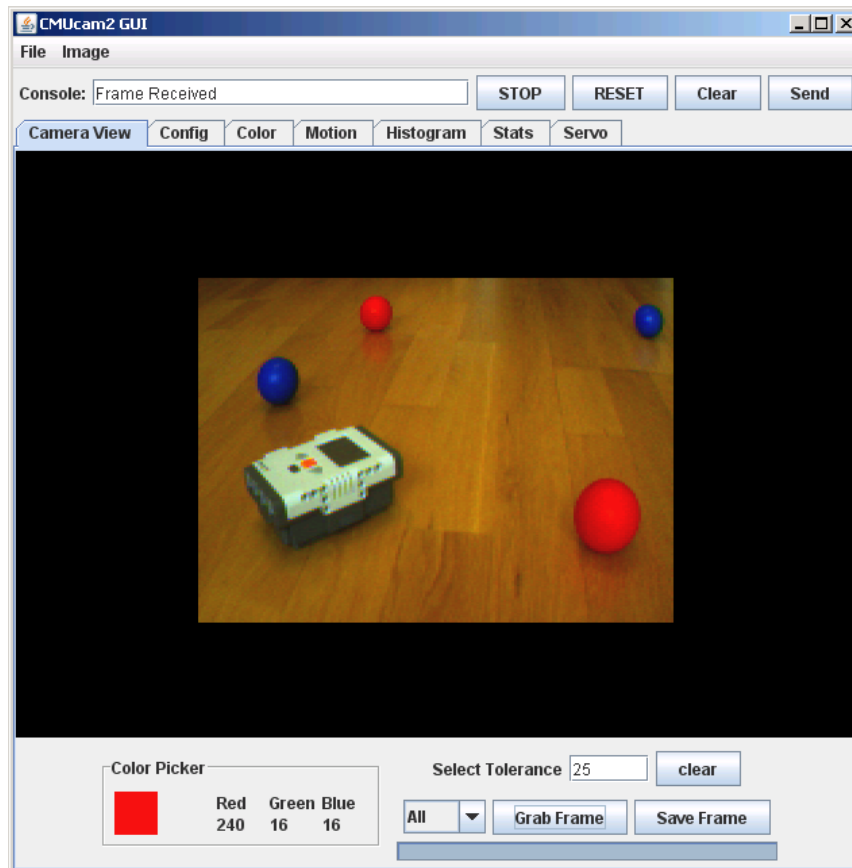

Figure 5: CMUcam2 GUI after acquiring a high resolution image. The mouse cursor was on top of the red ball in the foreground when taking the screenshot, resulting in the shown values in the "Color Picker" box.

What happens behind the scenes of CMUcam2 GUI is the following: In CMUcam2 emulation mode, the CMUcam3 listens on its serial port for new requests. Upon reception of a new request (sent by CMUcam2 GUI, for instance), it starts performing the requested operation and sends back according data. The CMUcam2 GUI reads these data and displays them accordingly.
Another mode of controlling the CMUcam3 in CMUcam2 emulation mode is, therefore, to use "hyperterminal" (or any corresponding program that is able to talk via a COM port). The list of available commands is available in the CMUcam2 documentation [5]. As an example, the firmware version of the camera can be obtained by sending "GV" (followed by a carriage return), see Figure 7. As another example, the command to start the color tracking (as shown in Figure 6) has the syntax "TC Rmin Rmax Gmin Gmax Bmin Bmax \r". Upon receiving this command, the CMUcam3 continuously sends "T packets" with the following syntax "T mx my x1 y1 x2 y2 pixels confidence\r" (see Figure 6), where the abbreviations have the following meaning:

mx	the middle of mass x value
my	the middle of mass y value
x1	the left most corner's x value
y1	the left most corner's y value
x2	the right most corner's x value
y2	the right most corner's y value
pixels	number of pixels in the tracked region, scaled and capped at 255: (pixels+4)/8
confidence	(number of pixels / area) * 256 of the bounded rectangle and capped at 255

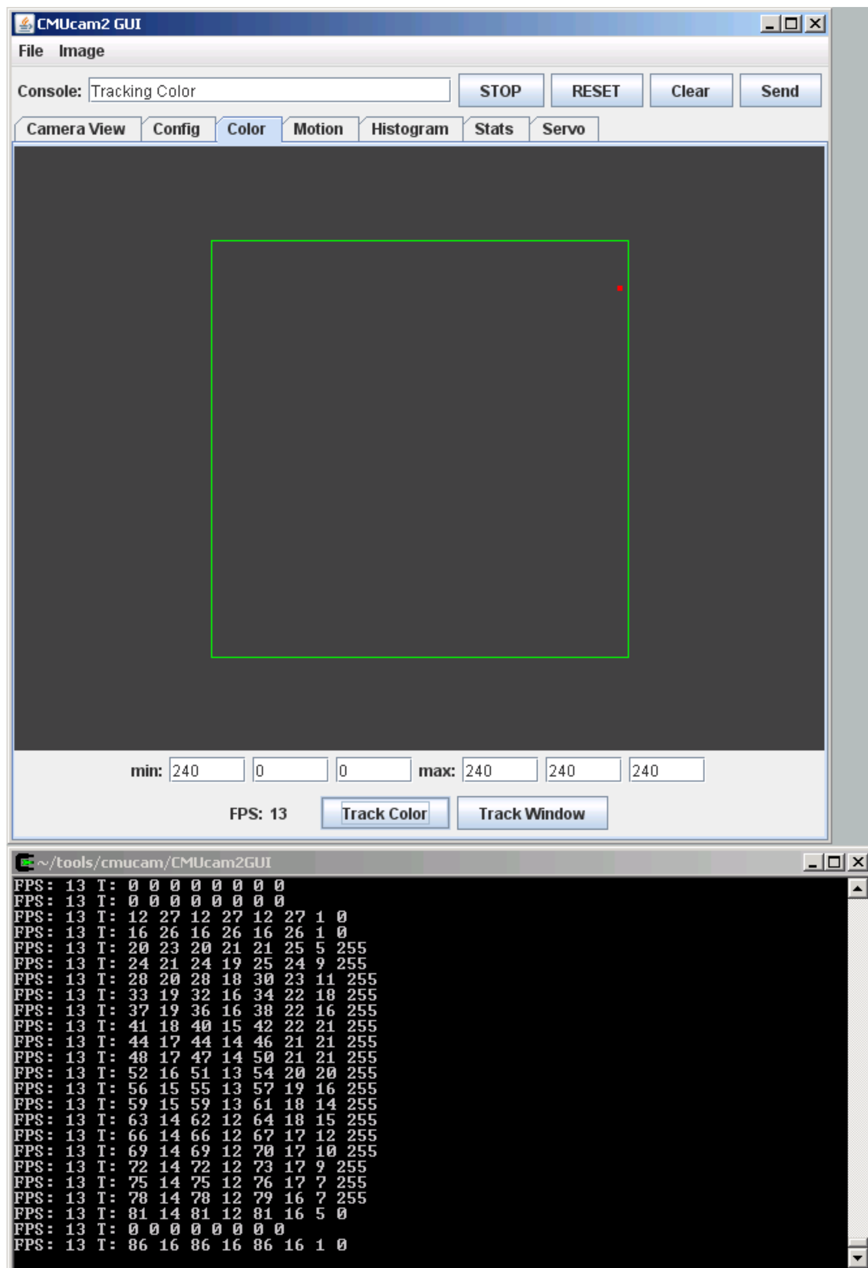

Figure 6: Tracking a red ball while rolling across the camera's field of vision from left to right. The min/max values specify bounds on the RGB values of the tracked object. FPS stands for frames per second, that is, how often the CMUcam3 sends updated coordinates. To obtain the textual output (lower window), CMUCam2 GUI needs to be started from a commandline window.

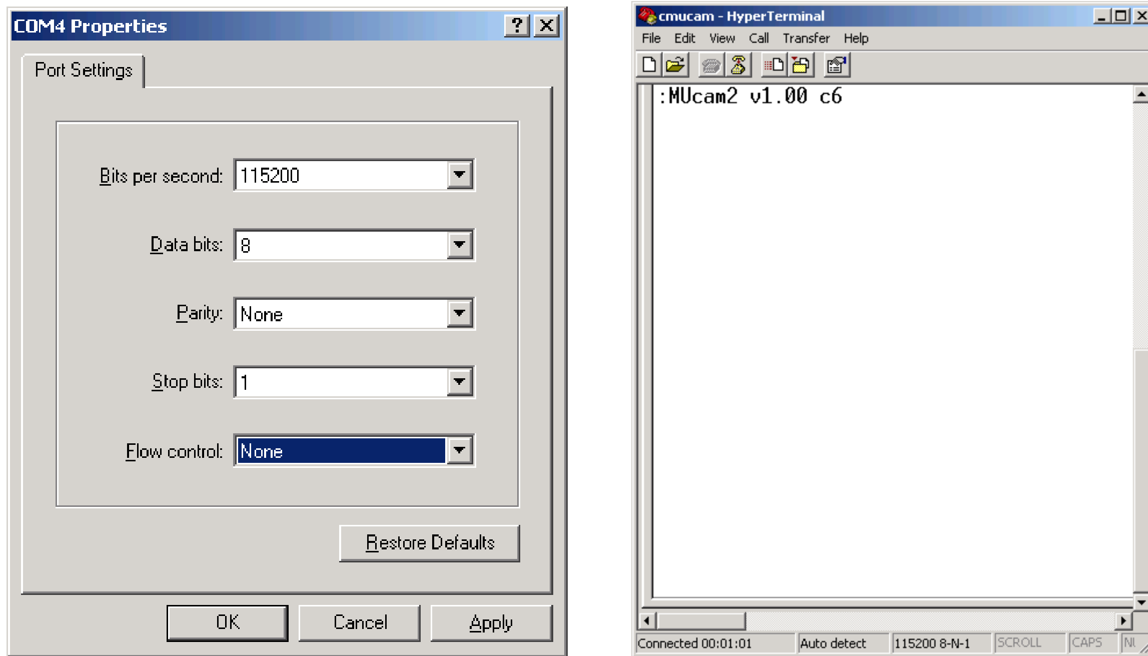

Figure 7: Settings for Hyperterminal and message received after switching on CMUCam. Messages sent by the CMUCam3 are terminated by "\r" (ASCII-code 13), resulting in a carriage return (but not a linebreak) in HyperTerminal. Following this linebreak, a ":" (colon) is displayed as a prompt.

Section 3

Coupling the CMUcam3 and the Lego Mindstorms NXT

To connect the NXT with the CMUcam3 an I^2C-to-RS232 bridge is used. This bridge is connected on the I^2C-side to one input port of the NXT and on the RS232-side to the RS232 connector of the CMUcam, see Figure 11. The used I^2C-to-RS232 bridge is the one described in the semester thesis by Leo den Hartog [7] (with a different software running on the bridge). A picture of the assembled bridge is shown in Figure 10. The system works as described in the following subsections and illustrated in Figure 8 and Figure 9. (The corresponding source code for the NXT and the I^2C-to-RS232 bridge is available in Appendix A and Appendix B.)

3.1 Communication from NXT to CMUcam3

To send data from the NXT to the CMUcam3, the NXT sends data to the I^2C-to-RS232 bridge (via I^2C) that forwards these data to the CMUcam3 (via RS232). This works as follows, see Figure 8. The NXT sends byte-wise commands to the I^2C-to-RS232 bridge. More specifically, to send a command (string), the NXT sends each character of the string separately by sending an I^2C address byte followed by one payload byte. After receiving a byte on the I^2C-to-RS232 bridge, it is immediately forwarded to the RS232 port (except if the byte has the value 128 or 129, see next section). This way, the NXT can easily send any command consisting of ASCII-characters (values between 0 and 127) to the CMUcam3.
As a possibility to check whether the byte was successfully received, the NXT can issue a receive request on the I^2C bus which will be answered by the I^2C-to-RS232 bridge by sending the byte which was received last.

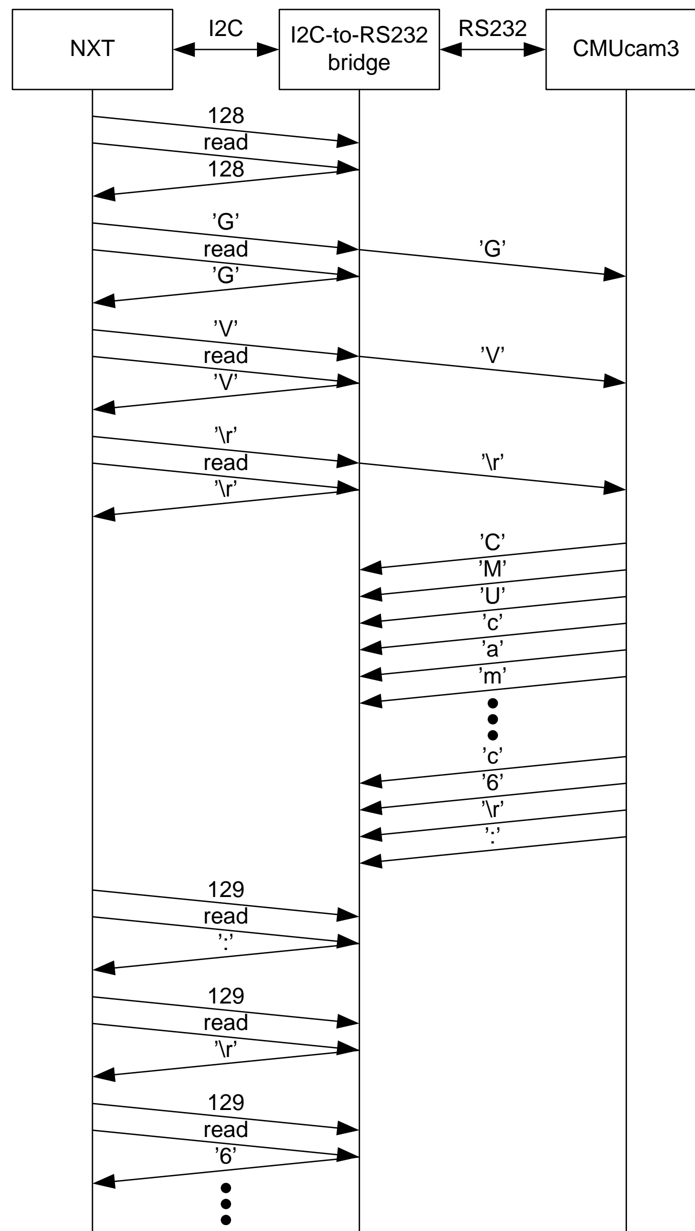

Figure 8: Sequence of transmissions for executing the get version (GV) command on the CMUcam3 and obtaining the result.

3.2 Communication from the CMUcam3 to the NXT

Upon reception of a command, the CMUcam3 starts to transmit the answer as soon as it is available. Since the data rate of the RS232 link (19200 bit/s) is higher than the one of the I^2C link (9600 bit/s), these data are buffered on the I^2C-to-RS232 bridge, where a ringbuffer of 89 bytes is implemented. Buffering on the ringbuffer is started when the I^2C-to-RS232 receives a byte with value 128 on the I^2C input (in which case the byte is not forwarded to the RS232 link). When the I^2C-to-RS232 bridge receives a byte with value 129 on the I^2C input, the buffering is stopped. By issuing a receive request on the I^2C bus after sending 129, the next byte from the ringbuffer will be returned (and not 129, as would be the case usually).
Refer to Figure 8 and Figure 9 for a graphical illustration of that protocol.

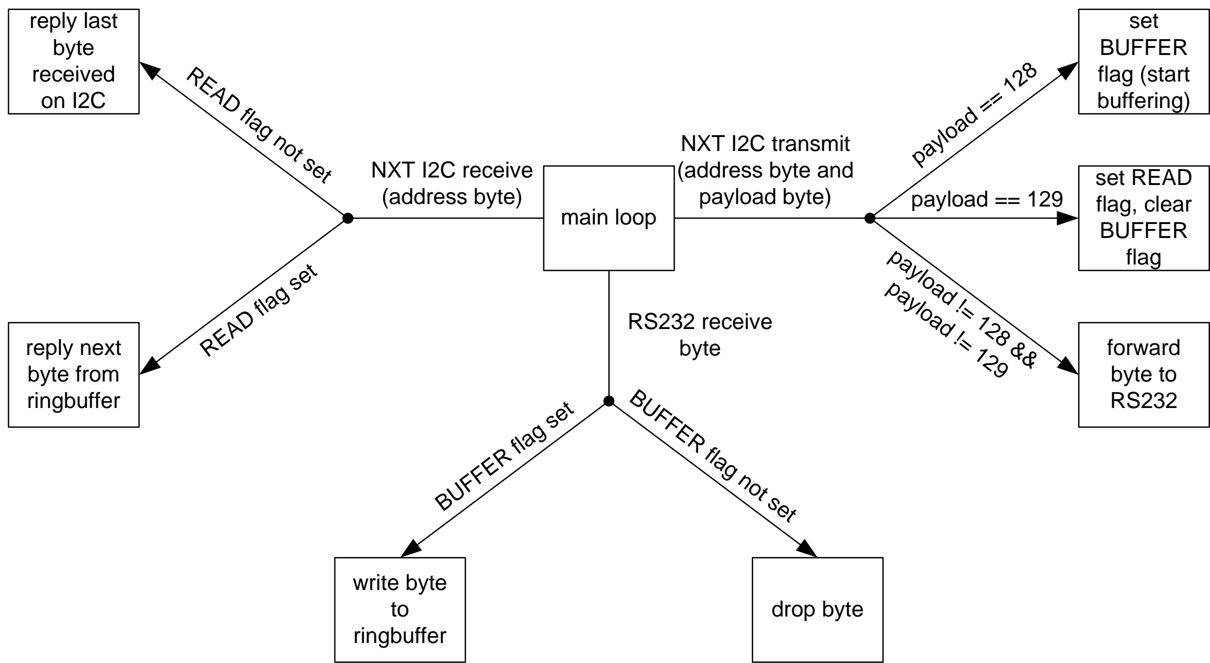

Figure 9: Overview of the behavior of the I^{2}C-to-RS232 bridge.

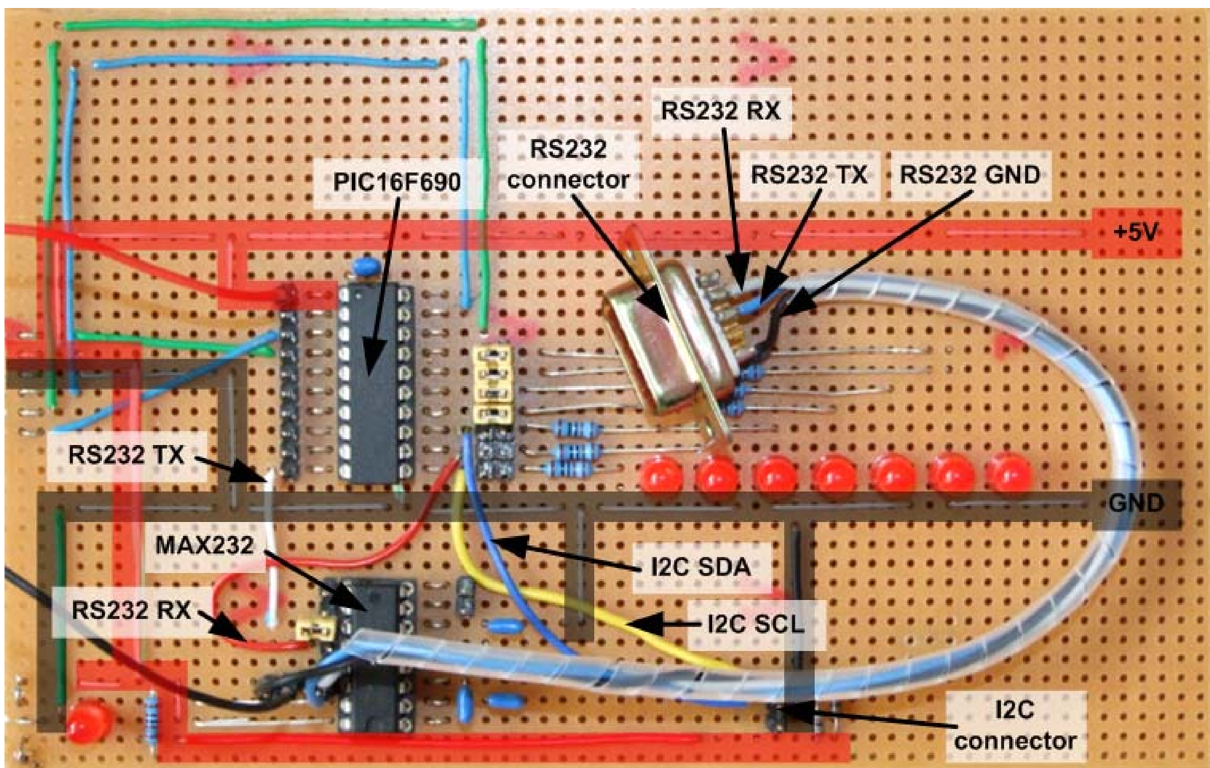

Figure 10: Top view of the I^{2}C-to-RS232 bridge.

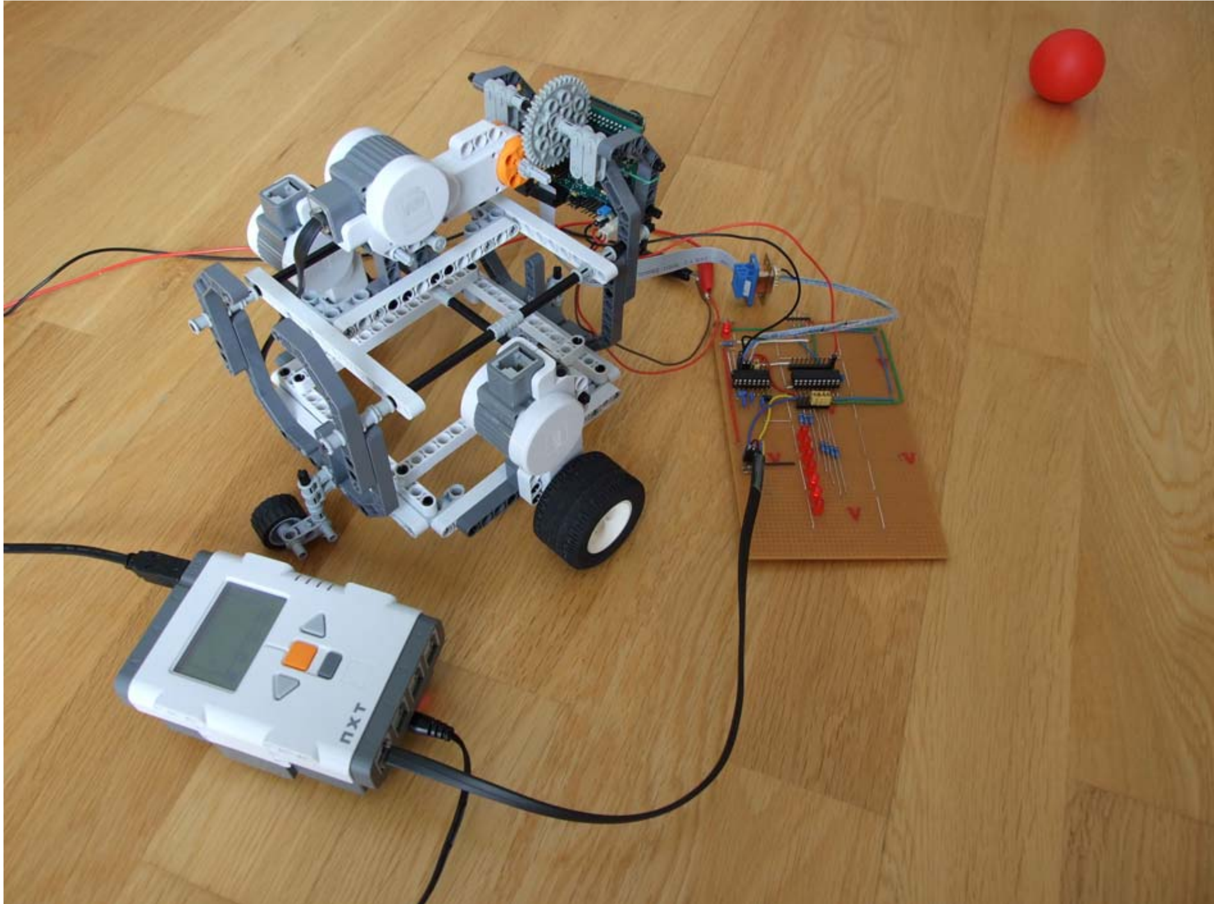

Figure 11: Coupling the CMUcam3 with the Lego Mindstorms NXT using the I^{2}C-to-RS232 bridge.

3.2 Implementation

The implementation of the described protocol is straightforward:

- **NXT:** Sending and receiving data via an I^{2}C port is described in the NXC programmer's guide [8]. Refer to Appendix A for a listing of the code.
- **CMUcam3:** On the CMUcam3 the standard CMUcam2 emulation firmware (19200 bit/s version) can be used without any modification.
- **I^{2}C-to-RS232 bridge:** The I^{2}C-to-RS232 bridge uses the PIC16F690 microcontroller to implement the desired functionality. Using the PIC16F690 data sheet [9] and two application notes [10] [11] for reference, the implementation is rather simple. Refer to Appendix B for a listing of the code.

3.3 Performance

Using the code in Appendix A and Appendix B, color tracking is possible with an update period of approximately 0.9 seconds. For most applications, this will be insufficient. Future work would be needed to remove the current bottleneck which is the gathering of data from the I^{2}C-to-RS232 bridge. This is due to three reasons:

- **I^{2}C data rate:** The data rate of the I^{2}C link is limited to 9600 bit/s. This cannot be changed, however.
- **Length of data packets:** The length of a "T packet" in the color tracking mode is up to 35 characters. The current implementation of the protocol on the NXT side always reads 69 bytes from the ringbuffer on the I^{2}C-to-RS232 bridge to make sure that at least one

complete packet is read. By introducing shorter packets (with less information, though), or by stopping reading new bytes from the I^2C-to-RS232 as soon as a complete package has been received the gathering of data could be accelerated.

- **Protocol:** Currently, for each received byte four bytes are transmitted over the I^2C link (address byte followed by 129 to indicate the request for data, another address byte to indicate readiness to receive data and the actual payload byte). This could be simplified to use the available bandwidth more efficiently.

Another, completely different approach would be to couple the CMUcam3 directly with the NXT over an I^2C link. In this case, an according driver for the I^2C interface on the CMUcam3 would need to be implemented. (At the time of writing this document, the ARM processor on the CMUcam3 could "only" be used as an I^2C master for writing data to MMD/SD memory cards. Slave mode has not been available.)

Appendix A

NXC Code for Lego Mindstorms NXT

```
/*************************************************************************
 * program to use CMUcam color tracking on NXT
 ************************************************************************/

#include "NXCDefs.h"

//application specific defines
#define I2C_PORT         IN_1
#define CHAR_WIDTH       6
#define LINE_WIDTH       16
#define SCREEN_X_ORIG    5
#define SCREEN_Y_ORIG    13
#define SCREEN_WIDTH     90
#define SCREEN_HEIGHT    50
#define SPEED            70

#define PIC_ADDRESS      38
#define START_BUF        128
#define GET_CHAR         129
#define BUF_SIZE         66

//CMUcam2 specific defines
#define CAM_RESET        "RS"
#define CAM_GET_VERSION  "GV"
#define CAM_TRACK_PARAMS "GT"
//TC Rmin Rmax Gmin Gmax Bmin Bmax \r
#define CAM_TRACK_COLOR  "TC 210 240 0 50 0 240"
#define CAM_LED_ON       "L0 1"
#define CAM_LED_OFF      "L0 0"

/**
 * send one byte to and receive one byte from i2c
 */
byte txrxI2C(const byte data) {
  byte recv[1];
  byte send[2];
  send[0] = PIC_ADDRESS;
  send[1] = data;
  while (I2CCheckStatus(I2C_PORT) != 0) {
  }
  I2CWrite(I2C_PORT, 1, send);
  while (I2CCheckStatus(I2C_PORT) != 0) {
  }
  while (I2CBytesReady(I2C_PORT) < 1) {
  }
  I2CRead(I2C_PORT, 1, recv);
  return recv[0];
}

/**
 * send a command byte-per-byte to i2c
 */
void sendCommandI2C(const string command) {
  int answer;
  txrxI2C(13);
  for (int i = 0; i < StrLen(command); i++) {
    answer = 0;
    do {
      answer = txrxI2C(command[i]);
    } while (answer != command[i]);
  }
  do {
    answer = txrxI2C(13);
  } while (answer != 13);
}

/**
 * send command to start buffering on the PIC16F690
 */
void startBufferI2C() {
  txrxI2C(START_BUF);
}
```

```
/**
 * send command to stop buffering on the PIC16F690
 */
void stopBufferI2C() {
  txrxI2C(GET_CHAR);
}

/**
 * read out the buffer from the PIC16F690
 */
void getResultI2C(byte &result[]) {
  for (int i = 0; i < BUF_SIZE; i++) {
    result[BUF_SIZE - i] = txrxI2C(GET_CHAR);
  }
}

/**
 * get a dummy result as expected from the PIC16F690
 */
void dummyGetResultI2C(byte &result[]) {
  for (int i = 0; i < BUF_SIZE; i++) {
    result[i] = 0;
  }
  string dummy = "T 10 20 255 4 55 6 77 128 T 10 20 30 40 abcd";
  for (int i = 0; i < StrLen(dummy); i++) {
   result[i] = dummy[i];
  }
  result[25] = 13;
}

/**
 * print the contents of the specified array
 */
void printResult(const byte result[]) {
  int i = 0;
  int j = 0;
  ClearScreen();
  for (i = 0; i < BUF_SIZE / LINE_WIDTH + 1; i++) {
    for (j = 0; j < LINE_WIDTH; j++) {
      if (j + i * LINE_WIDTH >= BUF_SIZE) {
        return;
      }
      byte help[1];
      help[0] = result[j + i * LINE_WIDTH];
      string temp;
      ByteArrayToStrEx(help, temp)
      switch (i) {
      case 0:
        TextOut(1 + j * CHAR_WIDTH, LCD_LINE1, temp, false);
        break;
      case 1:
        TextOut(1 + j * CHAR_WIDTH, LCD_LINE2, temp, false);
        break;
      case 2:
        TextOut(1 + j * CHAR_WIDTH, LCD_LINE3, temp, false);
        break;
      case 3:
        TextOut(1 + j * CHAR_WIDTH, LCD_LINE4, temp, false);
        break;
      case 4:
        TextOut(1 + j * CHAR_WIDTH, LCD_LINE5, temp, false);
        break;
      case 5:
        TextOut(1 + j * CHAR_WIDTH, LCD_LINE6, temp, false);
        break;
      case 6:
        TextOut(1 + j * CHAR_WIDTH, LCD_LINE7, temp, false);
        break;
      case 7:
        TextOut(1 +  j * CHAR_WIDTH, LCD_LINE8, temp, false);
        break;
      }
    }
  }
}
```

```
/**
 * extract the last complete answer from the given array. look for
 * the last carriage return (ASCII-code 13) in the array and return
 * the values preceding this carriage return until another carriage
 * return is found or the begin of the array is reached.
 */
void getLastAnswer(byte &result[]) {
  int i;
  int j;
  byte answer_temp[BUF_SIZE];
  answer_temp[0] = 0;

  //find last carriage return
  for (i = BUF_SIZE - 1; i > 0; i--) {
    if (result[i] == 13) {
      break;
    }
  }
  //no carriage return found
  if (result[i] != 13) {
    for (int j = 0; j < 8; j++) {
      return;
    }
  }

  //copy bytes until next carriage return or 0 to answer_temp
  j = 0;
  do {
    answer_temp[j++] = result[--i];
  } while (i > 0 && result[i - 1] != 13 && result[i - 1] != 0);
  for (i = 0; i <= j - 1; i++) {
    result[i] = answer_temp[j - 1 - i];
  }
  //copy bytes in reversed order to result
  for (i = j; i < BUF_SIZE; i++) {
    result[i] = 0;
  }
}

/**
 * reset the camera
 */
void resetCamera() {
  byte result[BUF_SIZE];
  do {
    sendCommandI2C(CAM_LED_ON);
    startBufferI2C();
    sendCommandI2C(CAM_RESET);
    sendCommandI2C(CAM_LED_OFF);
    getResultI2C(result);
    getLastAnswer(result);
  } while (result[0] != ':' || result[1] != 'A' ||
           result[2] != 'C' || result[3] != 'K')
}

/**
 * extract coordinates from a received result where a result is of the
 * form .* c0 c1 c2 c3 c4 c5 c6 c7\r.*, that is, a sequence of
 * integers separated by spaces, and terminated bya carriage return
 * (ASCII-code 13)
 */
void getCoordinates(const byte result[], byte &coords[]) {
  int i;
  //find last carriage return
  for (i = BUF_SIZE - 1; i > 0; i--) {
    if (result[i] == 13) {
      break;
    }
  }
  //no carriage return found
  if (result[i] != 13) {
    for (int k = 0; k < 8; k++) {
      coords[k] = 0;
      return;
    }
  }
  //process 8 integers (each integer has max. 3 characters)
  for (int j = 7; j >= 0; j--) {
```

```
    byte temp[3];
    int k;
    int l;
    string help_string = "";
    do {
      i--;
    } while (result[i] != ' ' && i > 0);
    if (i == 0) {
      for (int k = 0; k < 8; k++) {
        coords[k] = 0;
        return;
      }
    }
    l = i;
    k = 0;
    do {
      temp[k++] = result[++l];
    } while (result[l + 1] != 13 && result[l + 1] != ' ');
    ByteArrayToStrEx(temp, help_string)
    coords[j] = StrToNum(help_string);
  }
}

/**
 * print an array of 8 integers at the bottom two rows of the display
 */
void printCoordinates(const byte coords[]) {
  for (int i = 0; i < 8; i++) {
    NumOut(1,  LCD_LINE7, coords[0]);
    NumOut(25, LCD_LINE7, coords[1]);
    NumOut(50, LCD_LINE7, coords[2]);
    NumOut(75, LCD_LINE7, coords[3]);
    NumOut(1,  LCD_LINE8, coords[4]);
    NumOut(25, LCD_LINE8, coords[5]);
    NumOut(50, LCD_LINE8, coords[6]);
    NumOut(75, LCD_LINE8, coords[7]);
  }
}

/**
 * main. reset the camera and drive towards a colored object.
 */
task main() {
  int button_count;
  int counter = 0;
  byte result[BUF_SIZE];
  byte coords[8];
  bool tracking_enabled = false;

  SetSensorType(I2C_PORT, SENSOR_TYPE_LOWSPEED);
  SetSensorMode(I2C_PORT, IN_MODE_RAW);
  ResetSensor(I2C_PORT);
  TextOut(1, LCD_LINE1, "Trying to reset");
  TextOut(1, LCD_LINE2, "CMUcam2.");

  resetCamera();
  TextOut(1, LCD_LINE8, "Camera reset.");

  while(true) {
    if (ButtonCount(BTNLEFT, true) >= 1) {
      ClearScreen();
      TextOut(1, LCD_LINE1, "Left button.");
      startBufferI2C();
      sendCommandI2C(CAM_GET_VERSION);
      //sendCommandI2C(CAM_TRACK_PARAMS);
      Wait(200);
      getResultI2C(result);
      printResult(result);
      tracking_enabled = false;
    } else if (ButtonCount(BTNRIGHT, true) >= 1) {
      ClearScreen();
      TextOut(1, LCD_LINE1, "Right button.");
      resetCamera();
      TextOut(1, LCD_LINE8, "Camera reset.");
      tracking_enabled = false;
    } else if (ButtonCount(BTNCENTER, true) >= 0) {
      if (!tracking_enabled) {
        startBufferI2C();
```

```
      sendCommandI2C(CAM_TRACK_COLOR);
      Wait(200);
      tracking_enabled = true;
    }
    getResultI2C(result);
    startBufferI2C();

    ClearScreen();
    Wait(70);
    getCoordinates(result, coords);

    //draw a screen showing the position of the tracked object
    printCoordinates(coords);
    RectOut(SCREEN_X_ORIG, SCREEN_Y_ORIG, SCREEN_WIDTH, SCREEN_HEIGHT);
    CircleOut(SCREEN_X_ORIG + SCREEN_WIDTH - coords[0],
              SCREEN_Y_ORIG + SCREEN_HEIGHT - coords[1]/3, 2);
    RectOut(SCREEN_X_ORIG + SCREEN_WIDTH - coords[4],
            SCREEN_Y_ORIG + SCREEN_HEIGHT - coords[5]/3,
            (coords[4] - coords[2]), (coords[5] - coords[3])/3);

    //depending on the coordinates switch the motors on and off
    Off(OUT_AB);
    if (coords[0] == 0 && coords[3] != 0) {
      continue;
    }
    int x = coords[0];
    if (x < 40) {
      RotateMotor(OUT_A, SPEED, 3 * (x - 45));
      Off(OUT_A);
      RotateMotor(OUT_B, SPEED, 3 * (45 - x));
      Off(OUT_B);
    }
    else if (coords[0] > 50) {
      RotateMotor(OUT_A, SPEED, 3 * (x - 45));
      Off(OUT_A);
      RotateMotor(OUT_B, SPEED, 3 * (45 - x));
      Off(OUT_B);
    }
    OnRev(OUT_AB, 45 - abs(x - 45));
  }
 }
}
```

Appendix B

Assembly Code for PIC16F690

```
;i2c-to-rs232 bridge
;
;modification history:
;2008-09-08: created

#include <p16F690.inc>
  __config (_INTRC_OSC_NOCLKOUT & _WDT_OFF & _PWRTE_OFF & _MCLRE_OFF &
            _CP_OFF & _BOR_OFF & _IESO_OFF & _FCMEN_OFF)

#define NODE_ADDR  d'38'  ;i2c address
#define BUF_SIZE   d'89'  ;size of ringbuffer (limited by size of register
                          ;bank 0 (96 bytes) and the number of variables
                          ;allocated on register bank 0, that is, 7, see
                          ;cblock 0x20 below).
#define START_BUF  d'128' ;code for start buffering rs232 input
#define GET_CHAR   d'129' ;code for getting next byte from buffer
#define BUFFER     0      ;buffering flag (see variable state, below)
#define READ       1      ;read flag (see variable state, below)

  cblock 0x20             ;variables
i2c_rx                    ;buffer for one byte received from i2c
i2c_tx                    ;buffer for one byte to transmit over i2c
rs232_rx                  ;buffer for one byte received from rs232
rs232_tx                  ;buffer for one byte to transmit over r232
state                     ;state variable:
                          ;no bit set: bytes received from i2c are
                          ;  forwarded to rs232 and back-looped to i2c,
                          ;  bytes received from rs232 are ignored
                          ;bit 0 (BUFFER): set when rs232 input should be
                          ;  stored in ringbuffer, this is enabled by
                          ;  sending START_BUF on the i2c, the START_BUF
                          ;  byte itself is back-looped to i2c but not
                          ;  forwarded to rs232
                          ;bit 1 (READ): set when the next byte of the
                          ;  ringbuffer should be replied, this is enabled
                          ;  by sending GET_CHAR on the i2c, the GET_CHAR
                          ;  byte itself is not back-looped (but the byte
                          ;  of the ringbuffer, instead) and not forwarded
                          ;  to rs232
sspstat_masked            ;help variable for masking the SSPSTAT register
buf_ptr                   ;current position in the ringbuffer
buf                       ;ringbuffer (extends BUF_SIZE bytes towards the
                          ;end of register bank 0)
  endc

  ;set up all ports
  banksel PORTA
  clrf    PORTA
  clrf    PORTB
  clrf    PORTC

  ;port settings (1 is input, 0 is output)
  ;PORTA (8 pins) is unused except for pin 2
  movlw   b'00000000'
  banksel TRISA
  movwf   TRISA

  ;ports B (4 pins) are shared with ic2 and uart pins
  ;RB4 (pin 13): SDA (i2c)
  ;RB5 (pin 12): RX  (rs232)
  ;RB6 (pin 11): SCL (i2c)
  ;RB7 (pin 10): TX  (rs232)
  movlw   b'01110000'
  banksel TRISB
  movwf   TRISB

  ;PORTC (8 pins) is (partly) connected to LEDs
  movlw   b'00000000'
  banksel TRISC
  movwf   TRISC
```

```
  ;digital pins only
  banksel ANSEL
  movlw   h'0'
  movwf   ANSEL
  movlw   h'0'
  movwf   ANSELH
  clrf    CM1CON0
  clrf    CM2CON0
  banksel ADCON0
  clrf    ADCON0
  banksel ADCON1
  clrf    ADCON1

  ;disable interrupts
  banksel PIE1
  clrf    PIE1
  clrf    PIE2
  banksel INTCON
  clrf    INTCON
  clrf    PIR1
  clrf    PIR2

  ;uart settings: 19200 bits/s, 8 bit, no parity on transmission
  banksel SPBRGH
  movlw   d'0'
  movwf   SPBRGH
  movlw   d'12'
  movwf   SPBRG
  movlw   b'00100110'
  movwf   TXSTA

  movlw   b'10010000'     ;enable rs232, 8-bit continuous reception
  banksel RCSTA
  movwf   RCSTA

  ;i2c settings: slave mode, 7-bit address (the two assignments are
  ;necessary due to a silicon error on the PIC16F690, see PIC16F690
  ;data sheet errata)
  banksel SSPCON
  movlw   b'00111001'
  movwf   SSPCON
  movlw   b'00110110'
  movwf   SSPCON
  banksel SSPADD
  movlw   NODE_ADDR       ;set i2c address
  movwf   SSPADD

  ;configuration completed
  banksel PORTA
  clrf    PORTA
  clrf    PORTB
  clrf    PORTC

;wait for received byte on i2c
i2c_receive:
  bsf     PORTC,0
  bcf     PORTC,1
  bcf     PORTC,2
  btfss   PIR1,SSPIF
  goto    rs232_receive
  bcf     PIR1,SSPIF
  bcf     PORTC,0
  bsf     PORTC,2

i2c_rx_addr:
  banksel SSPSTAT
  movf    SSPSTAT,W
  bcf     STATUS,RP0
  andlw   b'00101101'
  movwf   sspstat_masked
  movlw   b'00001001'
  xorwf   sspstat_masked,W
  btfss   STATUS,Z
  goto    i2c_rx_data
  movf    SSPBUF,W        ;throw the address away (dummy read)
  goto    i2c_receive
```

```
i2c_rx_data:
  movlw   b'00101001'
  xorwf   sspstat_masked,W
  btfss   STATUS,Z
  goto    i2c_tx_addr
  movf    SSPBUF,W        ;put the data to i2c_rx, i2c_tx, and rs232_tx
  movwf   i2c_rx
  movwf   i2c_tx
  movwf   rs232_tx
  bcf     state,READ      ;clear reading flag
check_start_buf:
  movlw   START_BUF       ;check whether i2c_rx == START_BUF
  subwf   i2c_rx,W
  btfss   STATUS,Z
  goto    check_get_char
  call    initBuffer      ;clear buffer
  bsf     state,BUFFER    ;set buffering flag
  bsf     PORTA,2
  goto    i2c_receive
check_get_char:
  movlw   GET_CHAR        ;check whether i2c_rx == GET_CHAR
  subwf   i2c_rx,W
  btfss   STATUS,Z
  ;(i2c_rx != START_BUF && i2c_rx != GET_CHAR), so forward i2c_rx to rs232
  goto    rs232_transmit
  bcf     state,BUFFER    ;clear buffering flag
  bcf     PORTA,2
  bsf     state,READ      ;set reading flag
  goto    i2c_receive

i2c_tx_addr:
  banksel SSPSTAT
  movf    SSPSTAT,W
  bcf     STATUS,RP0
  andlw   b'00101100'
  movwf   sspstat_masked
  movlw   b'00001100'
  xorwf   sspstat_masked,W
  btfss   STATUS,Z
  goto    rs232_receive

i2c_transmit:
  banksel SSPSTAT
  btfsc   SSPSTAT,BF
  goto    i2c_transmit
  bcf     STATUS,RP0
  btfsc   state,READ
  call    readBuffer
write_byte:
  bcf     SSPCON,WCOL
  movfw   i2c_tx
  movwf   SSPBUF
  btfsc   SSPCON,WCOL
  goto    write_byte
  bsf     SSPCON,CKP      ;release the clock

rs232_receive:
  bcf     STATUS,RP0
  bcf     PORTC,0
  bsf     PORTC,1
  bcf     PORTC,2
  btfss   PIR1,RCIF
  goto    i2c_receive
  bcf     PORTC,1
  bsf     PORTC,2
  btfsc   RCSTA,OERR      ;catch overflow error
  goto    overflowerror
  btfsc   RCSTA,FERR      ;catch framing error
  goto    frameerror
  movfw   RCREG
  movwf   rs232_tx
  movwf   rs232_rx
  btfss   state,BUFFER
  ;replace the statement by 'goto rs232_transmit' to implement a
  ;rs232-backloop
  goto    i2c_receive
  call    writeBuffer
  goto    i2c_receive
```

```
overflowerror:
  bcf     RCSTA,CREN        ;pulse cren off
  movfw   RCREG             ;flush fifo
  movfw   RCREG             ;all three elements
  movfw   RCREG
  bsf     RCSTA,CREN        ;turn CREN back on, this clears OERR
  goto    i2c_receive
frameerror:
  movfw   RCREG             ;reading RCREG clears FERR
  goto    i2c_receive

rs232_transmit:
  btfss   PIR1,TXIF
  goto    rs232_transmit
  movfw   rs232_tx
  movwf   TXREG
  goto    i2c_receive


;initialize ringbuffer
initBuffer
  clrf    buf_ptr
clear_byte:
  movlw   buf               ;load address of buf into FSR
  addwf   buf_ptr,W
  movwf   FSR
  clrf    INDF
  movlw   d'1'
  addwf   buf_ptr,F
  movlw   BUF_SIZE
  subwf   buf_ptr,W         ;check whether (buf_ptr - BUF_SIZE) == 0
  btfss   STATUS,Z
  goto    clear_byte
  clrf    buf_ptr
  return

;write rs232_rx to current pointer position in ringbuffer
writeBuffer
  movlw   buf
  addwf   buf_ptr,W
  movwf   FSR
  movfw   rs232_rx
  movwf   INDF
  movlw   d'1'              ;increase pointer (wrap if it exceeds BUF_SIZE)
  addwf   buf_ptr,F
  movlw   BUF_SIZE          ;check whether buf_ptr < BUF_SIZE
  subwf   buf_ptr,W
  btfsc   STATUS,Z
  clrf    buf_ptr
  return

;read data at current pointer position from ringbuffer and put to i2c_tx
readBuffer
  movlw   d'1'              ;decrease pointer (wrap if it exceeds BUF_SIZE)
  subwf   buf_ptr,F
  movfw   buf_ptr
  xorlw   b'11111111'
  btfss   STATUS,Z
  goto    read_byte
  movlw   BUF_SIZE
  movwf   buf_ptr
  movlw   d'1'
  subwf   buf_ptr
read_byte:
  movlw   buf
  addwf   buf_ptr,W
  movwf   FSR
  movfw   INDF
  movwf   i2c_tx
  return
  end
```

References

[1] A. Rowe, A. Goode, D. Goel, C. Rosenberg and I. Nourbakhsh. CMUcam3: Open Source Programmable Embedded Color Vision Platform. http://www.cmucam.org

[2] A. Rowe, A. Goode, D. Goel, and I. Nourbakhsh. CMUcam3: An Open Programmable Embedded Vision Sensor. Technical Report CMU-RI-TR-07-13, Carnegie Mellon University, Robotics Institute, 2007. [online] http://www.ri.cmu.edu/pub_files/pub4/rowe_anthony_2007_1/rowe_anthony_2007_1.pdf

[3] CMUcam3 Datasheet. Carnegie Mellon University, Robotics Institute, 2007. [online] http://www.cmucam.org/attachment/wiki/Documentation/CMUcam3_datasheet.pdf

[4] CMUcam3 SDK Installation Guide. Carnegie Mellon University, Robotics Institute, 2007. [online] http://www.cmucam.org/attachment/wiki/Documentation/CMUcam3_sdk_guide.pdf

[5] CMUcam2 Vision Sensor – User Guide. Carnegie Mellon University, Robotics Institute, 2003. [online] http://www.cs.cmu.edu/~cmucam2/CMUcam2_manual.pdf

[6] CMUcam2 – Graphical User Interface. Carnegie Mellon University, Robotics Institute, 2003. [online] http://www.cs.cmu.edu/~cmucam2/CMUcam2GUI_overview.pdf

[7] L. den Hartog. Lego Mindstorms NXT Camera. Semester Thesis, ETH Zurich, Computer Engineering and Networks Laboratory, 2008.

[8] J. Hansen. Not eXactly C (NXC) Programmer's Guide, 2007. [online] http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_Guide.pdf

[9] PIC16F631/677/685/687/689/690 Data Sheet. Microchip Technology Inc., 2006.

[10] Application Note 774 – Asynchronous Communications with the PICmicro® USART. Microchip Technology Inc., 2008.

[11] Application Note 734 – Using the PIC Devices' SSP and MSSP Modules for Slave I^2C^{TM} Communication. Microchip Technology Inc., 2008.