**Appendix A: Reports**

* Report: Remote Control Car Teardown
* Report: Handyboard Serial Port

**Report: Remote Control Car Teardown**

Wireless Mesh Network

Peter Fyon 100652096

peter.fyon@gmail.com

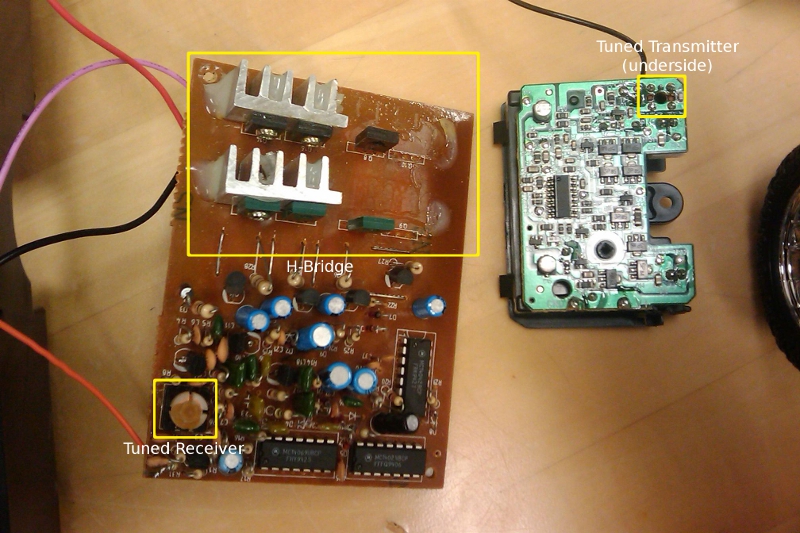
1.0 Introduction

The purpose of this paper is to report on our findings from tearing down one skid-steer (SS) and one turn-steer (TS) remote control car. The report will cover the method used in taking apart the vehicles, the hardware at a general level, and the important lessons learned from the process.

2.0 Method

The deconstruction of each vehicle was simple. The plastic casing was removed with a Phillips head screwdriver to expose the control board inside. Tests were performed on the motors by connecting one to a motor port on the handyboard and using a program, written in Interactive C, to vary the rotational speed (using pulse width modulation) of the motor in both directions. It was noted that the LEDs on the board did not dim visibly with motors from either car, and concluded that they were not drawing more current than the board could provide.

3.0 Hardware

Figure 1: Car Driver Boards. Skid steering car on left, turn steering car on right. Note that two power transistors and two heatsinks were removed from the skid steering car board.

3.1 Skid Steer Car

* + Uses a variety of analog components to filter the incoming signal before sending it to the control circuitry
  + Uses two independently controllable motors on each side of the vehicle to control steering and forward/reverse at the same time
  + Motors are supplied approximately 12 volts from eight AA cell batteries.
  + Each motor is driven forwards or backwards with their own H bridge made up of four heat-sunk power transistors.
  + Turning while moving requires fine control of the rotation speed of the motors
  + The car is capable of turning 360˚ while remaining in place

3.2 Turn Steer Car

* + Uses much fewer analog components, and more integrated circuits than the SS car.
  + Uses one motor to drive the rear wheels and one motor to turn the front wheels left or right.
  + Motors are supplied approximately 3 volts from two AA cell batteries.
  + The default position for the front wheels is straight.
  + Turning the front wheels is limited to all left or all right.

Both of the remote control cars use a radio tuned to the same frequency as the transmitter in the controller. Also, both vehicles have a small capacitor shorting the motor's terminals, presumably to filter out high-frequency noise caused by the motors.

4.0 Lessons

We learned the following things during the deconstruction of this vehicle:

1. SS vehicles are less precise in their steering than TS vehicles as the rate of turn is related to the friction between each drive wheel and the ground. The friction may not be the same for each wheel due to different ground materials, differing wear on each wheel, etc.
2. The motors are simple to control with pulse width modulation (PWM), and can be driven directly from the Handyboard without additional circuitry.
3. The turn angle for our TS vehicle can be made more precise by replacing the motor responsible for turning with a servo and some modifications with the gearing ratio between the servo and the axle.
4. Both controllers used simple microswitches to detect forward/reverse and left/right directives from the controller.

These lessons may influence the design for our controller and steering.

**Report: Handyboard Serial Port**

Wireless Mesh Network

John Koh 100684909

johnnykoh.is@gmail.com

1.0 Introduction

The purpose of this paper is to report on our findings on the serial port and communication method used on the Handyboards. This report will cover the hardware functionality of the serial port on the Handyboard.

2.0 Analysis

In order to communicate with the Handyboard, the compiler must connect through the serial interface. By analyzing the hardware in between the serial interface and the Handyboard, the Handyboard can be configured to work with the XBee node.

3.0 Technical Specifications

See the attached diagram on the CPU and Memory Circuit Schematic.(page 37 in manual)

The Handyboard only has one pair of receive (RxD) and transmit (TxD) but uses an RJ14 plug, which normally houses two pairs. The following table is how the RJ14 plug is laid out on the Handyboard.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Position** | **RJ14 Pin** | **Pair** | **Tip/Ring** | +/- | **Colours (wire, Handyboard-side)** |
| 1 |  |  |  |  |  |
| 2 | 1 | 2 | Tip | + | Black |
| 3 | 2 | 1 | Ring | - | Red |
| 4 | 3 | 1 | Tip | + | Green |
| 5 | 4 | 2 | Ring | - | Yellow |
| 6 |  |  |  |  |  |

**Appendix B: Initial Designs**

* Handy Board Communication Protocol Version 1

**Report: Handyboard Communication Protocol**

Wireless Mesh Network

Peter Fyon 100652096

peter.fyon@gmail.com

John Koh 100684909

johnnykoh.is@gmail.com

1.0 Introduction

The purpose of this paper is to define a standard protocol for communication between the controller handyboard and the car drive handyboard. The document will cover the functional requirements and technical implementation details.

2.0 Requirements

The following requirements were defined during the requirements elicitation process:

1. The controller will not send conflicting signals to the vehicle (Left and right, or forward and backwards).
2. The controller will be able to send both forward or backwards, and directional commands at the same time.
3. The controller will be able to centre the directional wheels.
4. If there is no command to send, the controller will send an opcode representing “do nothing”.
5. There will not be a significantly noticeable delay between when an input is supplied to the controller and when the corresponding action is taken by the vehicle.
6. The controller will send a packet approximately every 50 milliseconds.[1](" \l "sdfootnote1sym)
7. The vehicle will consume a packet approximately every 50 milliseconds.

3.0 Technical Specifications

The packet structure will be arranged as follows:

* Single byte packet containing opcode
  + Low five bits for forwards/backwards, left/right, and centre commands.
  + High 3 bits reserved for future use.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Reserved** | **Reserved** | **Reserved** | **Centre** | **Forward** | **Backward** | **Left** | **Right** |  |
| Centre Wheels: | | | | | | | | | |
|  | **X** | **X** | **X** | **1** | **X** | **X** | **0** | **0** |  |
| Do-Nothing | | | | | | | | | |
|  | **X** | **X** | **X** | **0** | **0** | **0** | **0** | **0** |  |
| Forward: | | | | | | | | | |
|  | **X** | **X** | **X** | **X** | **1** | **0** | **X** | **X** |  |
| Reverse: | | | | | | | | | |
|  | **X** | **X** | **X** | **X** | **0** | **1** | **X** | **X** |  |
| Left: | | | | | | | | | |
|  | **X** | **X** | **X** | **0** | **X** | **X** | **1** | **0** |  |
| Right: | | | | | | | | | |
|  | **X** | **X** | **X** | **0** | **X** | **X** | **0** | **1** |  |

Note that while the controller may send multiple commands in the same packet, it is the responsibility of the controller to ensure that each command does not conflict with any others.

3.1 Encoding

Encoding is done on the controller. Every directional switch will add their corresponding instruction to the instruction packet. After collecting inputs, the controller will send the instruction packet to the R/C car.

3.2 Decoding

Decoding is done on the R/C car. The R/C car will decode the instructions by performing a bit-wise AND with the instruction packet and each possible instruction, except Do-Nothing. When a bit-wise AND returns true, the corresponding instruction will be performed. Once all the possible instructions are checked, the car will sleep.

[1](" \l "sdfootnote1anc)Through experimental testing, it was determined that an action taken by the vehicle every 50 ms (such as moving forward or backwards) was indistinguishable from continuous motion.

**Appendix C: Code**

* Vehicle Drive Software, Unreliable Protocol
* Vehicle Control Software, Unreliable Protocol
* Common Header
* Serial Interrupt Routine
* Serial Helper Functions
* OpCode Test Software
* Serial Test Software, Receiving
* Serial Test Software, Sending
* Vehicle Drive Software, Reliable Protocol
* Vehicle Control Software, Reliable Protocol
* Protocol Function Definitions

car\_drive-noack.ic

/\*

\* car\_drive is a program which receives commands over the UART

\* to drive motors and (possibly in a later version) a servo.

\* It is intended to control a remote control car over a mesh

\* network provided by ZigBee capable chips that transmit commands

\* from a controller to the vehicle running this program.

\*

\* @author Peter Fyon

\*

\*/

#include sci\_isr.c

#include header.ic

// Motor number definitions

#define MOTORDRIVE 0 // fwd/reverse drive motor

#define MOTORTURN 1 // turn motor

// Motor speed definitions

#define FWDSPEED 50 // Speed to set forward motor

#define REVSPEED -50 // Speed to set reverse motor

#define STOPSPEED 0

#define LEFTSPEED 120

#define RIGHTSPEED -120

#define DEBUGGING 1

//Arbitrary number to send to kill processes

#define KILLPROCESS 99

//direction

// 0 - stop

// 1 - forward

// -1 - reverse

int direction = 0;

void goForward(){direction = 1;}

void goReverse(){direction = -1;}

void stop(){direction = 0;}

//leftRight

// 0 - centre wheels

// 1 - right

// -1 - left

int leftRight = 0;

void turnRight(){leftRight = 1;}

void turnLeft(){leftRight = -1;}

void centreWheels(){leftRight = 0;}

void main()

{

//Initialize code variable

int code = DONOTHING;

int forward = 0;

int reverse = 0;

int left = 0;

int right = 0;

int centre = 0;

int counter = 0;

//Variables to store process IDs of the motor processes

int driveProcess = start\_process(driveProcess());

int turnProcess = start\_process(turnProcess());

//Disable infra red decoding since we're not using it

disableIRDecoding();

//Disable control of the UART by interactive C and set it up to use interrupts

initSerial();

//Loop

while(1)

{

while(!dataAvailable())

{

defer();

counter++;

if (counter == 40){

direction = 0;

leftRight = 0;

counter = 0;

}

}

counter = 0;

code = serialGetChar();

forward = code & FORWARD;

reverse = code & REVERSE;

left = code & LEFT;

right = code & RIGHT;

centre = code & CENTRE;

//if(DEBUGGING) printf("\ncode: %b",code);

//If the opcode is DONOTHING, stop moving

if(code != DONOTHING)

{

//BEGIN forward/reverse code

if((forward > 0) && (reverse == 0))

{

//Go forward

goForward();

} else if((reverse > 0) && (forward == 0))

{

//Go reverse

goReverse();

}else{

stop();

}

//END forward/reverse code

//BEGIN left/right/centre code

if((left > 0) && (right == 0))

{

//Turn left

turnLeft();

} else if((right > 0) && (left == 0))

{

//Turn right

turnRight();

}

//Need to use an if instead of else if because interactive C can't handle

// more than one else if in a row...apparently.

//Centre opcode takes precedence over the other signals

//if(centre > 0)

//END left/right/centre code

} else{

//Do nothing

stop();

leftRight = 0;

}

//Sleep for 45ms

// msleep(50L);

}

//Code currently cannot reach here

direction = KILLPROCESS;

leftRight = KILLPROCESS;

}

void driveProcess()

{

while(direction != KILLPROCESS)

{

//off(MOTORDRIVE);

if(direction == 1)

{

//Go forward

motor(MOTORDRIVE,FWDSPEED);

}

if(direction == 0)

{

//Stop

off(MOTORDRIVE);

}

if(direction == -1)

{

//Go reverse

motor(MOTORDRIVE,REVSPEED);

}

//Reset instruction to 0

//direction = 0;

// msleep(50L);

}

}

void turnProcess()

{

while(leftRight != KILLPROCESS)

{

if(leftRight == 1)

{

//Turn right

motor(MOTORTURN,RIGHTSPEED);

}

if(leftRight == 0)

{

//Centre wheels

off(MOTORTURN);

}

if(leftRight == -1)

{

//Turn left

motor(MOTORTURN,LEFTSPEED);

}

//msleep(50L);

}

}

controller-noack.ic

/\*

\* controller is a program which sets the code upon receiving

\* instructions from the microswitch signals from the

\* Handyboard.

\*

\* @author John Koh

\* @date February 19, 2010

\*

\*/

#include "header.ic"

#include sci\_isr.c

//switch definitions, direction switch\_number

#define go\_forward 15

#define go\_back 14

#define go\_left 13

#define go\_right 12

#define center 11

void main() {

int code = DONOTHING;

int dnflag = 0;

initSerial();

disableIRDecoding();

while (1) {

if (digital(go\_forward)) {

code = code + FORWARD;

}

if (digital(go\_back)) {

if ((digital(go\_forward)) ||

(code & REVERSE == REVERSE) ||

(code & FORWARD == FORWARD)) {

code = code;

} else {

code = code + REVERSE;

}

}

if (digital(go\_left)) {

code = code + LEFT;

}

else if (digital(go\_right)) {

code = code + RIGHT;

}

if (digital(center)) {

code = CENTRE;

}

printf("%d \n",code);

if ((code == DONOTHING) && (dnflag == 1)) {

}else if (code == DONOTHING){

dnflag = 1;

serialPutChar(code);

}else{

serialPutChar(code);

dnflag = 0;

}

msleep(100L);

code = DONOTHING;

}

}

header.ic

/\*

\* This is a header file with various constants defined for use by car\_drive and car\_control programs.

\*

\* @author Peter Fyon

\* @date February 5, 2010

\*

\*/

//Define opcodes

//Included in the comments are the required bits and the 'don't cares'

//Be sure you don't send conflicting opcodes

#define FORWARD 0x08 // xxxx 10xx

#define REVERSE 0x04 // xxxx 01xx

#define LEFT 0x02 // xxxx xx10

#define RIGHT 0x01 // xxxx xx01

#define CENTRE 0x10 // xxx1 xx00

#define DONOTHING 0x00 // xxx0 0000

//IR decoding takes up 11% of the processor when it's enabled (which is default)

void disableIRDecoding()

{

bit\_clear(0x39, 0b00000010);

}

void enableIRDecoding()

{

bit\_set(0x39, 0b00000010);

}

sci\_isr.asm

/\* This sets up the serial interrupt service routine \*/

/\* First Version: Written by Anton L. Wirsch 20 Nov 1997 \*/

/\* Second Version: Written by Tim Gold 27 May 1998

BYU Robotics Lab

goldt@et.byu.edu

Really, the only thing left from the original code are a few

lines in the .asm file. Everything else I pretty much had to

rewrite from scratch to get it to work the way I wanted to.

But the orignal code by Anton was a very helpful starting point.

Needed files: serial\_isr.c

serial\_isr.icb

serial\_isr.asm (needed to change the buffer size)

The buffer size here is 32 bytes (probably much larger than it needs

to be.) To change the buffer size, do the following:

1. Change the BUFFER\_SIZE constant in serial\_isr.c to the

desired number of bytes.

2. Edit the line in this fils which contains

the word "EDIT" in the comment so that the value

matches that of BUFFER\_SIZE.

3. Recreate the serial\_isr.icb file by typing the following:

> as11\_ic serial\_isr.asm

\*/

/\* change this line to match your library path... \*/

#include "6811regs.asm"

ORG MAIN\_START

variable\_CURRENT:

FDB 00 \* ptr to next data to be read by user

variable\_INCOMING:

FDB 00 \* number of bytes received (circular count)

variable\_BASE\_ADDR:

FDB 00 \* base address of buffer (to be set by init routine)

variable\_DATA\_FLAG:

FDB 00 \* flag set when data is available

variable\_buffer\_ptr:

FDB 00 \* pointer to CURRENT buffer

subroutine\_initialize\_module:

/\* change this line to match your library path... \*/

#include "ldxibase.asm"

ldd SCIINT,X

std interrupt\_code\_exit+1

ldd #interrupt\_code\_start

std SCIINT,X

rts

interrupt\_code\_start:

ldad variable\_INCOMING \* store INCOMING into AB

cmpb #00 \* compare B with 0

bhi skip \* goto "skip" if (B > 0)

ldx variable\_BASE\_ADDR \* STORE ADDRESS OF ARRY IN X

inx \* SKIP THE FIRST (?)

inx \* TWO BYTES (?)

inx \* OFFSET TO THE HIGHER BYTE (?)

stx variable\_buffer\_ptr \* SAVE PTR VALUE

bra cont

skip:

ldx variable\_buffer\_ptr \* load buffer pointer into x

cont:

ldad variable\_INCOMING \* load INCOMING into AB

incb \* increment INCOMING

cmpb #4 \* compare B and 4 --EDIT TO CHANGE BUFFER SIZE--

beq reset\_count \* if a=32, goto reset\_count

bra cont1

reset\_count:

ldad #00 \* set count to zero

cont1:

stad variable\_INCOMING \* store AB into INCOMING

ldab SCSR \* load SCSR (SCI status register) into B (why?)

ldab SCDR \* load SCSR (SCI data register) into B

stab ,X \* store data in array

inx \* increment by two bytes

inx

stx variable\_buffer\_ptr \* save the pointer value

ldad #01 \* load 1 into AB

stad variable\_DATA\_FLAG \* store AB into DATA\_FLAG (indicating data is available)

interrupt\_code\_exit:

jmp $0000

sci\_isr.c

/\* C program to read serial port with interrupt service routine \*/

/\* First version: Written by Anton Wirsch 20 Nov 1997 \*/

/\*

Second Version: Written by Tim Gold 27 May 1998

BYU Robotics Lab

goldt@et.byu.edu

Really, the only thing left from the original code are a few

lines in the .asm file. Everything else I pretty much had to

rewrite from scratch to get it to work the way I wanted to.

But the orignal code by Anton was a very helpful starting point.

Needed files: serial\_isr.c

serial\_isr.icb

serial\_isr.asm (needed to change the buffer size)

The buffer size here is 32 bytes (probably much larger than it needs

to be.) To change the buffer size, do the following:

1. Change the BUFFER\_SIZE constant below to the

desired number of bytes.

2. Edit the line(s) in the serial\_isr.asm which contain

the word "EDIT" in the comment so that the value

matches that of BUFFER\_SIZE.

3. Recreate the serial\_isr.icb file by typing the following:

> as11\_ic serial\_isr.asm

\*/

//Peter Fyon, Feb 5, 2010: Changed filename from serial\_isr.c to sci\_isr.c as some

// older programs have issues with filenames greater than 8 characterss

#define BUFFER\_SIZE 4 /\* change buffer size here -- see above \*/

/\* various constants used by the program... \*/

#define BAUD 0x102b /\* baud rate set to 9600 \*/

#define SCCR2 0x102d

#define SCCR1 0x102c

#define SCSR 0x102e

#define SCDR 0x102f

#use "SCI\_ISR.ICB"

int buffer[BUFFER\_SIZE]; /\* this is the actual buffer \*/

int return\_char;

void initSerial()

{

/\* Call this routine to activate the serial interrupt handler. \*/

int i,temp;

/\* clear out buffer \*/

for(i=0; i<BUFFER\_SIZE; i++)

{

buffer[i] = 0;

}

/\* clear vairous flags \*/

DATA\_FLAG = 0;

INCOMING = 0;

CURRENT = 0;

return\_char = 0;

/\* pass address of buffer to interrupt routine \*/

buffer\_ptr = (int) buffer;

BASE\_ADDR = (int) buffer;

/\* activate interrupt routine \*/

temp = peek(SCCR2);

temp |= 0x24;

poke(SCCR2, temp);

poke(0x3c, 1);

}

void closeSerial()

{

int temp;

/\* deactivate the interrupt routine \*/

temp = peek(SCCR2);

temp &= 0xdf;

poke(SCCR2, temp);

//Peter Fyon: Not sure why this line is here, READ\_SERIAL doesn't even exist in the asm routine

//READ\_SERIAL = 0x0000;

poke(0x3c, 0);

}

void serialPutChar(int c)

{

/\* call this function to write a character to the serial port \*/

while (!(peek(0x102e) & 0x80));

poke(0x102f, c);

}

int dataAvailable()

{

/\* This function can be used to check to see if any data is available \*/

return DATA\_FLAG;

}

int serialGetChar()

{

/\* Create blocking getchar for serial port... \*/

/\* loop until data is available \*/

while(!DATA\_FLAG){};

/\* get the character to return \*/

return\_char = buffer[CURRENT];

/\* check for wrap around... \*/

CURRENT++;

if(CURRENT == BUFFER\_SIZE)

CURRENT = 0;

if(CURRENT == INCOMING)

DATA\_FLAG = 0;

return return\_char;

}

opcode\_test.ic

/\*

\* Program to test the opcodes for car\_drive over the serial port.

\*/

#include sci\_isr.c

#include constants.ic

void main()

{

int val;

int opcode[] = {CENTRE, FORWARD, REVERSE, LEFT, RIGHT, DONOTHING};

int i = 0;

initSerial();

while(1)

{

if(start\_button())

{

val = opcode[i];

i++;

if(i == 6)

{

i = 0;

}

}

if(stop\_button())

{

serialPutChar(val);

printf("\nSent: %d", val);

msleep(50L);

}

}

/\*

//for(i = 0; i < 6; i++)

while(i < 6)

{

val = opcode[i];

serialPutChar(val);

printf("\nSent: %d", val);

sleep(0.2);

while(!start\_button()){}

i++;

}

i = 0;

\*/

}

serial\_test\_receive.ic

/\*

\* Program to test the assembly serial interrupt routine on the handyboard.

\*/

#include sci\_isr.c

void main()

{

int receivedChar = 0;

initSerial();

printf("\nStarting");

while(!stop\_button())

{

receivedChar = serialGetChar();

printf("\nReceived: %d",receivedChar);

}

printf("\nStopping");

closeSerial();

}

serial\_test\_send.ic

/\*

\* Program to test the assembly serial interrupt routine on the handyboard.

\*/

#include sci\_isr.c

void main()

{

int val = 0;

initSerial();

printf("\nStarting");

while(!stop\_button())

{

if(start\_button())

{

val = knob();

serialPutChar(val);

printf("\nSent: %d", val);

sleep(0.2);

}

}

printf("\nStopping");

closeSerial();

}

car\_drive.ic

/\*

\* car\_drive is a program which receives commands over the UART

\* to drive motors and (possibly in a later version) a servo.

\* It is intended to control a remote control car over a mesh

\* network provided by ZigBee capable chips that transmit commands

\* from a controller to the vehicle running this program.

\*

\* @author Peter Fyon

\*

\*/

#include sci\_isr.c

#include header.ic

#include protocol.ic

// Motor number definitions

#define MOTORDRIVE 0 // fwd/reverse drive motor

#define MOTORTURN 1 // turn motor

// Motor speed definitions

#define FWDSPEED 75 // Speed to set forward motor

#define REVSPEED -75 // Speed to set reverse motor

#define STOPSPEED 0

#define LEFTSPEED 30

#define RIGHTSPEED -30

#define SLEEP\_DURATION 50L

#define DEBUGGING 1

//Arbitrary number to send to kill processes

#define KILLPROCESS 99

//direction

// 0 - stop

// 1 - forward

// -1 - reverse

int direction = 0;

void goForward(){direction = 1;}

void goReverse(){direction = -1;}

void stop(){direction = 0;}

//leftRight

// 0 - centre wheels

// 1 - right

// -1 - left

int leftRight = 0;

void turnRight(){leftRight = 1;}

void turnLeft(){leftRight = -1;}

void centreWheels(){leftRight = 0;}

void main()

{

//Initialize code variable

int code = DONOTHING;

int forward = 0;

int reverse = 0;

int left = 0;

int right = 0;

int centre = 0;

int lastSeqNum = 0;

//Variables to store process IDs of the motor processes

int driveProcess = start\_process(driveProcess());

int turnProcess = start\_process(turnProcess());

//Disable infra red decoding since we're not using it

disableIRDecoding();

//Disable control of the UART by interactive C and set it up to use interrupts

initSerial();

if(DEBUGGING)

{

printf("\nStarted");

}

//Loop

while(1)

{

while(!dataAvailable())

{

defer();

}

code = serialGetChar();

if(DEBUGGING) printf("\n%d : %d",code, getSeqNum(code));

if(matchesParity(code) && getSeqNum(code) != lastSeqNum)

{

//Packet's parity is correct and seq num is the next expected one

forward = code & FORWARD;

reverse = code & REVERSE;

left = code & LEFT;

right = code & RIGHT;

centre = code & CENTRE;

//Alternate sequence number

lastSeqNum++;

lastSeqNum = lastSeqNum % 2;

//Send an ACK for the packet received

sendAck(lastSeqNum);

printf("Ack %d",lastSeqNum);

if(DEBUGGING) printf("\ncode: %b",code);

//If the opcode is DONOTHING, stop moving

if(code != DONOTHING)

{

//BEGIN forward/reverse code

if((forward > 0) && (reverse == 0))

{

//Go forward

goForward();

} else if((reverse > 0) && (forward == 0))

{

//Go reverse

goReverse();

}

//END forward/reverse code

//BEGIN left/right/centre code

if((left > 0) && (right == 0))

{

//Turn left

turnLeft();

} else if((right > 0) && (left == 0))

{

//Turn right

turnRight();

}

//Need to use an if instead of else if because interactive C can't handle

// more than one else if in a row...apparently.

//Centre opcode takes precedence over the other signals

if(centre > 0)

{

//Centre wheels

centreWheels();

}

//END left/right/centre code

} else

{

//Do nothing

stop();

}

//Sleep for 45ms

//msleep(SLEEP\_DURATION);

} else

{

//Sequence number is out of order or packet is somehow mangled

//Reply with an ACK for the last successfully received packet

sendAck(lastSeqNum);

printf("Ack %d",lastSeqNum);

}

}

//Code currently cannot reach here

direction = KILLPROCESS;

leftRight = KILLPROCESS;

}

void driveProcess()

{

while(direction != KILLPROCESS)

{

if(direction == 1)

{

//Go forward

motor(MOTORDRIVE,FWDSPEED);

}

if(direction == 0)

{

//Stop

off(MOTORDRIVE);

}

if(direction == -1)

{

//Go reverse

motor(MOTORDRIVE,REVSPEED);

}

//Reset instruction to 0

//direction = 0;

msleep(SLEEP\_DURATION);

}

}

void turnProcess()

{

while(leftRight != KILLPROCESS)

{

if(leftRight == 1)

{

//Turn right

motor(MOTORTURN,RIGHTSPEED);

}

if(leftRight == 0)

{

//Centre wheels

off(MOTORTURN);

}

if(leftRight == -1)

{

//Turn left

motor(MOTORTURN,LEFTSPEED);

}

msleep(SLEEP\_DURATION);

}

}

void sendAck(int seqNum)

{

//Create an ACK packet

int packet = createPacket(ACK, seqNum, 0);

serialPutChar(packet); }

controller.ic

/\*

\* controller is a program which sets the code upon receiving

\* instructions from the microswitch signals from the

\* Handyboard.

\*

\* @author John Koh

\* @date February 19, 2010

\*

\*/

#include header.ic

#include protocol.ic

#include sci\_isr.c

//switch definitions, direction switch\_number

#define go\_forward 15

#define go\_back 14

#define go\_left 13

#define go\_right 12

#define center 11

void main() {

int code = DONOTHING;

int sequenceNo = 1;

int receivedPacket = 0;

initSerial();

disableIRDecoding();

while (1)

{

if (digital(go\_forward)) {

code = code + FORWARD;

}

if (digital(go\_back)) {

if ((digital(go\_forward)) ||

(code & REVERSE == REVERSE) ||

(code & FORWARD == FORWARD)) {

code = code;

} else {

code = code + REVERSE;

}

}

if (digital(go\_left)) {

code = code + LEFT;

}

else if (digital(go\_right)) {

code = code + RIGHT;

}

if (digital(center)) {

code = CENTRE;

}

while(1)

{

printf("\n%d",code);

serialPutChar(createPacket(DATA, sequenceNo, code));

msleep(100L);

if(dataAvailable())

{

receivedPacket = serialGetChar();

if (getSeqNum(receivedPacket) == sequenceNo)

{

printf(",%d",receivedPacket);

//Alternate the sequence number

sequenceNo++;

sequenceNo = sequenceNo % 2;

code = DONOTHING;

break;

}

}

}

}

}

protocol.ic

/\*

\* This file defines the protocol and provides methods to create and access contents of packets

\*

\*/

//Default ACK packet for receiver: X1X00000

//Default data packet:X0XXXXXX

//Parity bit is calculated by adding together all the other bits in the packet

#define paritybit 0b10000000

//Used for the receiver to send ACKs back

//1 for an ACK, 0 for a NAK

//Set to ACK if it's a data packet

#define acknakbit 0b01000000

//Alternating sequence bit, initially 0

#define seqnumbit 0b00100000

#define databit0 0b00000001

#define databit1 0b00000010

#define databit2 0b00000100

#define databit3 0b00001000

#define databit4 0b00010000

#define NAK 0

#define ACK 1

#define DATA 1

//Returns true if the packet matches the parity bit, or false if it doesn't

int matchesParity(int packet)

{

int bitcount = 0;

int parity = 0;

//Get the parity bit

if(packet & paritybit)

{

parity = 1;

}

if(packet & acknakbit)

{

bitcount++;

}

if(packet & seqnumbit)

{

bitcount++;

}

if(packet & databit0)

{

bitcount++;

}

if(packet & databit1)

{

bitcount++;

}

if(packet & databit2)

{

bitcount++;

}

if(packet & databit3)

{

bitcount++;

}

if(packet & databit4)

{

bitcount++;

}

if(bitcount % 2 == parity)

{

return 1;

} else

{

return 0;

}

}

int createPacket(int ack, int seqNum, int data)

{

int packet = 0;

int parity;

//Set the data bits

packet |= data;

//Set the bit if it's an ACK/data packet

if(ack)

{

packet |= acknakbit;

}

//Set the seqnum bit to whatever it should be

if(seqNum)

{

packet |= seqnumbit;

}

parity = createParity(ack, seqNum, data);

if(parity)

{

packet |= paritybit;

}

}

/\*

\* createParity takes the ack/nak, sequence number, and data bits and

\* adds them together, returns the result mod 2 for parity bit

\*/

int createParity(int ackNak, int seqNum, int data)

{

int temp = 0;

int i;

int bitcount = 0;

bitcount += ackNak;

bitcount += seqNum;

//Add up the data bits

for(i = 0; i < 5; i++)

{

temp = data << 1;

if(temp)

{

bitcount++;

}

}

return (bitcount % 2);

}

int getSeqNum(int packet)

{

if(packet & seqnumbit)

{

return 1;

}

return 0;

}

//The opposite is true, meaning !isAck() means it's a NAK

int isAck(int packet)

{

if(packet & acknakbit)

{

return 1;

}

return 0;

}