Individual Lab Report 5

Guillermo Manuel Cidre

Team G – Bob's builders

Teammates: Eric Newhall, Michael O’Connor, Christian Heaney-Secord

IRL05

3/19/15

Individual Work:

For this week, I improved upon the code style of the motor control handling for the arduino. Initially, I would add in a new motor handling unto the arduino by copying and pasting code of another motor handling. But since we are probably dealing with a total of at least 5 motors, it would seem impractical to follow this style. So, I completely changed the code by lableling the repeated variables for each copied code into an array and implementing a dynamic for loop, which adjust to number of motors available, to handle each individual motor. Ultimately, this is suppose to shorten the size of code and make adding new motor to the code easier by just requiring the programmer to expand few arrays instead of copying and pasting a large chunk of code.

In addition, I attempted to implement the state diagram for the part placer unto the motor handling code. For the demo, we used Eric's state diagram code instead of the code I did for simplicity. It is more simple because my motor handling code with the state diagram code is actually a round robin multitask program while Eric's demo code is not multitasking. I also added a calibration state unto the program which makes sure every component is in the right place. The new motor handling code is on figure 1 of this paper. The multitasking flow of the new code is on figure 2.

Challenges / Issues:

The main challenge that I had was implementing states on the arduino motor handling code. The reason for this is mainly that it was impossible to simply implement the ideal plan of the state diagram. The round robin multitasking introduced in the code prevented the states the ability of controlling the motors directly. So I had to implement several arrays, which updated in the motor handling phase, that gives information about the motors themselves like if it finished moving or not. And control the motors indirectly by modifying what the state would want to do based on the information from the arrays. Another issue was that when program enter a state, it will hold previous information from arrays and not the current ones. So basically, if the state code demanded the motor to move and checked if it finished moving, it will say it stopped moving because the motor stopped moving before given the demand. This was circumvented by performing the demand when the state transition or ensured to check when the information was updated.

Team Work:

Michael O'Connor and Christian Heaney-Secord worked on assembling the flux. They also attached the wire cutter tool that would be used for the actual wire cutting. They also installed the flux dispensor unto the system. Our arduino ran out of pins. So no wiring was done.

Eric got the computer vision on the playstation EYE working and implemented the simplified state diagram of the part placer unto the arduino with image orientation checking

program on the raspberry pi.

Future Plans:

Hopefully, we can get access to an arduino mega in order to be able to add in the motors of the flux dispensor. I plan to add in those motors into the arduino motor handling programs. In addition, I plan to finish and debug the addition of state diagram unto the code. Possibly, I'll help contribute in the wire cutting.

```
 //variables for the three Stepper Motor
float IRDistance[] = {0.0, 0.0}; //scale version of Value[]
const int stepPin[] = {6, 4};
const int dirPin[] = {5, 10};
const int enPin[] = {9, 12};
int toggle[] = {0, 0};
float stepDistance[] = {0.0, 0.0};
boolean stepDone[] = {false, false}; //indicate if a stepper motor finished moving

//For our DC motors
int DCMotorPinA[] = {7};
int DCMotorPinB[] = {8};

//etc
unsigned char psw = 0;
const int limitsw = 13;
const int elePin = 2;
int ele = 0;

unsigned int j = 0; //our temporary variable for selecting proper index
unsigned int len; //length of iterating arrays

unsigned char serialValue = 0;
float Value[] = {0, 0}; //stepper motor value
unsigned char DCValue[] = {1};
boolean Write[] = {false, false};
boolean DCWrite[] = {false};
boolean Wrote = false; //Check if system wrote value

//For calibration
boolean calibrated = false; //must calibrate all pieces
unsigned char cState = 0;
```

```
boolean cDone = true; //indicates if all motors have moved to the proper place

//Part Placer State setup
unsigned char ppState = 0; //GO_TO_PICKUP
unsigned char image_orientation = 0; //Invalid value to prevent state from transitioning too
early
boolean get_orientation = true; //must ensure the orientation is saved throughout the ppStates
const float pickup_loc = 50;
const float drop_loc = 0;
const float tray_drop = 0;

//For time keeping
unsigned long time; //in millisecond
const unsigned long motordelay = 300; //The time for motor to reach bottom



void setup() {
  Serial.begin(9600);
  while (!Serial) { //Copy
   ; // wait for serial port to connect. Needed for Leonardo only
 }
  len = sizeof(stepPin)/sizeof(int);
  for(j=0; j<len; j++)
  {
    pinMode(stepPin[j],OUTPUT);
    pinMode(dirPin[j],OUTPUT);
    pinMode(enPin[j],OUTPUT);
  }

  pinMode(limitsw, INPUT);
  pinMode(elePin, OUTPUT);


  len = sizeof(DCMotorPinA);
  for(j=0; j<len; j++)
  {
    pinMode(DCMotorPinA[j], OUTPUT);
    pinMode(DCMotorPinB[j], OUTPUT);
  }
}
```

```
void loop() {
  if(Serial.available()) //Read value and decide what to do with it
  {
    serialValue = Serial.read();

//Code for GUI: Do not Delete.

//  Wrote = updateWrite();
//      if( (Wrote == 0) && ( 100 > serialValue) &&  (serialValue >= 90) )
//      {
//        j = serialValue - 90; //get current index
//        Write[j] = true;
//      }
//      else if( (Wrote == false) && ( 110 > serialValue) &&  (serialValue >= 100) )
//      {
//        j = serialValue - 100; //get current index
//        DCWrite[j] = true;
//      }
//      else if( (Wrote == false) && ( (serialValue) == 50))
//      {
//        ele = (!ele) & 0x01;
//      }

//Code for raspberry communication.(can only have one ON at a time)
    if(get_orientation)
    {
      image_orientation = serialValue - 30; //converts char number to number
      get_orientation = false;
    }
  } //attributed sensor readings

  //Reading any other essential inputs
  psw = digitalRead(limitsw);
  //Implementation of calibration

  if(calibrated == false)
  {
    //perform calibration
    switch(cState)
    {
      case 0: //pull DC up
        DCValue[0] = 0; //Must verify this!!!!
```

```
  if(psw == 1) //Must Verify this!!!
  {
    DCValue[0] = 1; //Stop the DC
    cState = 1; //transition to next step
    //set new coordinates
    len = sizeof(stepPin)/sizeof(int);
    for(j=0; j<len; j++)
    {
      Value[j] = 255;
    }
  }
  break;

case 1: //Stepper goes to 255
  len = sizeof(stepPin)/sizeof(int);
  cDone = true;
  for(j=0; j<len; j++)
  {
    cDone = cDone && stepDone[j];
  }
  if(cDone) //all components are finished moving
  {
    cState = 2;
    len = sizeof(stepPin)/sizeof(int);
    for(j=0; j<len; j++)
    {
      Value[j] = 0;
    }
  }
  break;

case 2: //Stepper goes to 0
  len = sizeof(stepPin)/sizeof(int);
  cDone = true;
  for(j=0; j<len; j++)
  {
    cDone = cDone && stepDone[j];
  }
  calibrated = cDone; //we are done!
  break;
```

```
    }
  }
  else
  {
    //implement all states here
    //using cDone as a temporary variable on boolean checks

    //implementation of ppState
    switch(ppState)
    {
      case 0: //GO_TO_PICKUP
        cDone = (Value[1] == pickup_loc);
        Value[1] = pickup_loc;
        if(cDone && stepDone[1] && (image_orientation != 0))
        { //the stepDone array has been updated and signals that the stepper finished moving
and we got an orientation
          ppState = 1; //T2.1
          time = millis(); //reset timer
        }
        break;

      case 1: //EXTEND
        //implement timing directions
        DCValue[0] = 2; //should go down
        if(millis() - time > motordelay)
        {
          DCValue[0] = 1; //stop
          ppState = 2;
        }
        break;

      case 2: //RETRACT
        ele = 1; //turn on magnet
        DCValue[0] = 0; //Ensure this is DC up
        if(psw == 1) //Must Verify this!!!
        {
          DCValue[0] = 1; //Stop the DC
          if(image_orientation == 1 || image_orientation == 3)
          {
            ppState = 4; //Go to Drop off
          }
          else if(image_orientation == 2 || image_orientation == 4)
```

```
        {
          ppState = 3;//perform twist
        }
      }
      break;

    case 3: //TWIST
      //implement timing directions
      DCValue[0] = 2; //should go down
      if(millis() - time > motordelay)
      {
        DCValue[0] = 1; //stop
        ppState = 4;
      }
      break;

    case 4: //GO_TO_DROP_OFF(This section needs to account tray position TODO)
      cDone = (Value[1] == drop_loc);
      Value[1] = drop_loc;
      Value[0] = tray_drop; //ensure
      if(cDone && stepDone[1] && stepDone[0] && (image_orientation != 0))
      { //the stepDone array has been updated and signals that the stepper finished moving
and we got an orientation
        ppState = 5; //T2.6
      }
      break;



    case 5: //DROP_PART(copy of extend state
      //implement timing
      ppState = 6;
      break;
    case 6: //RESET(copy of retract)
      ele = 0; //turn off magnet
      DCValue[0] = 0; //Ensure this is DC up
      if(psw == 1) //Must Verify this!!!
      {
        DCValue[0] = 1; //Stop the DC
        ppState = 0;
      }
      break;
```

```
  }
}
```

```
//Update each motor properly

//Stepper Motor controlled by IR sensor
len = sizeof(stepPin)/sizeof(int);
for(j=0; j<len; j++)
{
  IRDistance[j] = (Value[j]*2.5)/(255*2.5); //approx distance in cm
  //Motor 1
  if(stepDistance[j] < IRDistance[j]-.005 && IRDistance[j] < 2.0){
     digitalWrite(enPin[j], LOW);
     digitalWrite(dirPin[j], HIGH);
     toggle[j] = (!toggle[j]) & 0x01;
     digitalWrite(stepPin[j], toggle[j]);
     stepDistance[j] += .001;
     stepDone[j] = false;
  }else if(stepDistance[j] > IRDistance[j]+.005 && IRDistance[j] < 2.0){
    digitalWrite(enPin[j], LOW);
    digitalWrite(dirPin[j], LOW);
    toggle[j] = (!toggle[j]) & 0x01;
    digitalWrite(stepPin[j], toggle[j]);
    stepDistance[j] -= .001;
    stepDone[j] = false;
  }
  else {
    digitalWrite(enPin[j], HIGH);
    stepDone[j] = true;
  }
}

//DC Motor Implementation
len = sizeof(DCWrite)/sizeof(boolean);
for(j=0; j<len; j++)
{
  switch(DCValue[j]) {
    case 0:
```

```
        //turn left
        digitalWrite(DCMotorPinA[j],LOW);
        digitalWrite(DCMotorPinB[j],HIGH);
        break;
      case 1:
        digitalWrite(DCMotorPinA[j],LOW);
        digitalWrite(DCMotorPinB[j],LOW);
        break;
      case 2:
        digitalWrite(DCMotorPinA[j],HIGH);
        digitalWrite(DCMotorPinB[j],LOW);

    }
  }


  // Write Pin
  digitalWrite(elePin, ele);

  delay(1); //For fast speed


  Serial.flush();
}


//boolean updateWrite()
//{
//    //Stepper Case
//    len = sizeof(Write)/sizeof(boolean);
//    for(j=0; j<len; j++)
//    {
//      if(Write[j])
//      { //update value return true
//        Value[j] = serialValue;
//        Write[j] = false;
//        return true;
//      }
//    }
//    //DC Motor Case
//    len = sizeof(DCWrite)/sizeof(boolean);
//    for(j=0; j<len; j++)
```

```
//    {
//      if(DCWrite[j])
//      { //update value return true
//        DCValue[j] = serialValue;
//        DCWrite[j] = false;
//        return true;
//      }
//    }
//
//    return false; //No writing required
//}
```

Figure 1: the new code with GUI code exempted

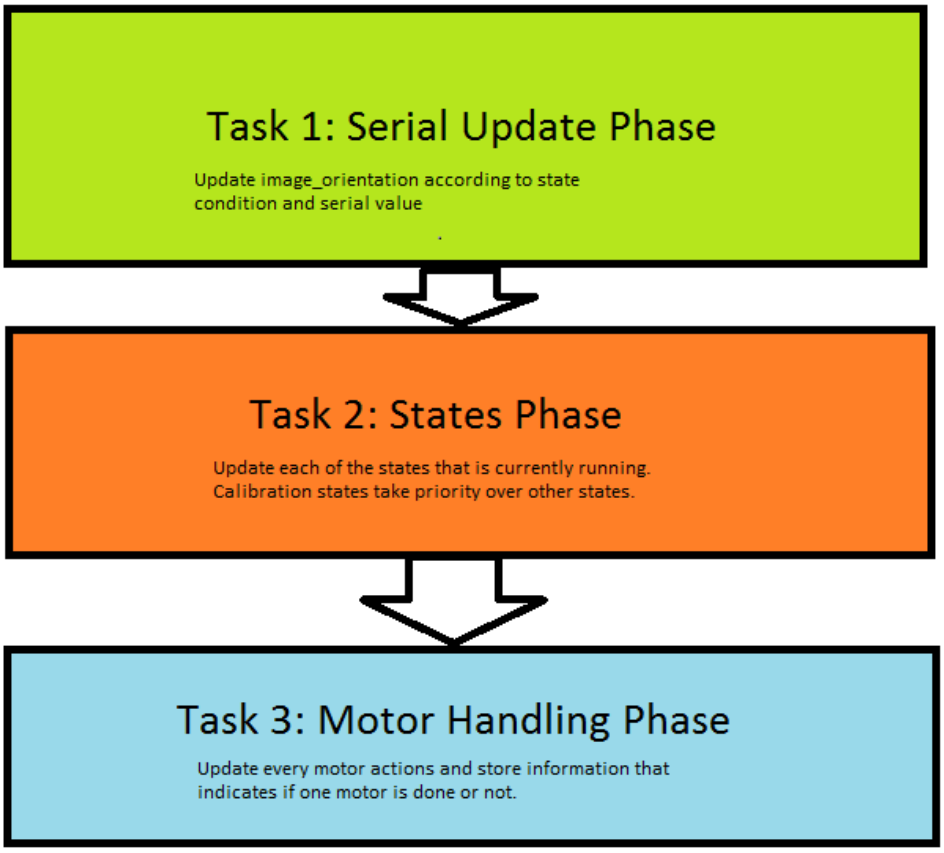

Figure 2: the round robin multitasking flow of the main program.