Appendix B: Arduino Processing Source: push_keys.pde

```
/*
  Nikolas Wolfe
  9/18/09

  Push Key Program

  This program utilizes the output pins of an Arduino
  Duemilanove to electronically press keys on the keypad
  of a cellular phone.
*/

/*=====================================================================
  PIN ASSIGNMENTS

  The variable names below correspond to pins linked to electrical nodes
  on the cellular keypad. In order to press a given key, the switches
  controlling the corresponding vertical and horizontal nodes must be
  BOTH closed.

  Logically the keypad is arranged thus:

  Node:   1   2   3

        | 1 | 2 | 3 |   A
        -------------
        | 4 | 5 | 6 |   B
        -------------
        | 7 | 8 | 9 |   C
        -------------
        | * | 0 | # |   D

  These numbered I/O pins are the only thing that may have to be modified
  if implementing this project on another board. In terms of the logical
  value of, say, key 1, it is up to the person implementing the circuit
  to ensure that the connection between Node 1 and Node A produces a 1.

  Other pins numbered here are the ACCEPT pin, which corresponds to a
  node required to push a button that accepts an incoming call. Also,
  the PWR and INCOMING_CALL variables, respectively, represent the power
  button and the pin driving the incoming call interrupt.

  =====================================================================
*/
#include <ControllerTransferProtocol.h>

/*=====================================================================
  THESE MAY CHANGE...
*/
const int INCOMING_CALL  = 2;
const int NODE_1         = 3;
const int NODE_2         = 4;
const int NODE_3         = 5;
const int NODE_A         = 6;
const int NODE_B         = 7;
const int NODE_C         = 8;
const int NODE_D         = 9;
const int ACCEPT         = 10;
const int PWR            = 11;
const int LED_PIN        = 13;

// System Voltage
const int SYSTEM_VOLTAGE = 12;

// Multiplier for Cut-Off Values
const double CO_MULTIPLIER = 0.992;

// Logging interval (# of 30-second periods between logs)
```

```
const int LOG_INTERVAL = 1; // once every 30 seconds

/*=====================================================================
  DO NOT MODIFY ANYTHING BELOW HERE....
*/
const int ZERO[]         = { NODE_2, NODE_D };
const int ONE[]          = { NODE_1, NODE_A };
const int TWO[]          = { NODE_2, NODE_A };
const int THREE[]        = { NODE_3, NODE_A };
const int FOUR[]         = { NODE_1, NODE_B };
const int FIVE[]         = { NODE_2, NODE_B };
const int SIX[]          = { NODE_3, NODE_B };
const int SEVEN[]        = { NODE_1, NODE_C };
const int EIGHT[]        = { NODE_2, NODE_C };
const int NINE[]         = { NODE_3, NODE_C };

/*=====================================================================
  INT ARRAY FOR NUMBER PAD

  This array can be used anywhere to obtain the pin numbers of a given
  key using an index value. This is useful when translating data inputs into
  output values. The values for # and * are not included here.
  =====================================================================
*/
const int* decimalArray[] = {  ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE };

/*=====================================================================
  OTHER VARIABLES
  =====================================================================
*/
const int STAR[]        = { NODE_1, NODE_D };
const int POUND[]       = { NODE_3, NODE_D };
const int START_CALL[]  = { ACCEPT, ACCEPT };
const int POWER[]       = { PWR, PWR };

const int THIRTY_SECONDS = 100;
const int interruptZero = 0;

int LOOP_COUNTER = 0;

boolean executeDataTransfer = false;
boolean isRun = false;

/*=====================================================================
  DIAGNOSTIC PARAMETERS

  These are the doubles which polled for transfer
  =====================================================================
*/

double panelVoltage = 0;
double panelAmperage = 0;
double batteryVoltage = 0;
double batteryAmperage = 0;
double totalKilowattHours = 0;

// Cut-Off Multiplier for Voltage
double LOW_VOLTAGE_CUTOFF = (SYSTEM_VOLTAGE * CO_MULTIPLIER);

/*=====================================================================
  CURRENT DATA FRAME

  This array is populated by polling the controller for these values over
  a serial transfer protocol.
  =====================================================================
*/
double valArr[] = {
  panelVoltage,
  panelAmperage,
  batteryVoltage,
  batteryAmperage,
```

```
  totalKilowattHours
};

/*========================================================================
  LOGGER DATA FRAMES

  These are the data frames that store historical data from the previous
  two logging intervals. They are initialized to zero until their time
  elapses.
  ========================================================================
*/
double valArrMinusOne[] = { 0, 0, 0, 0, 0 };
double valArrMinusTwo[] = { 0, 0, 0, 0, 0 };

/* size of the data frame */
const int NUM_DIAGNOSTIC_PARAMS = 5;

/*========================================================================
  ALERT PHONE NUMBER

  This array contains the phone number of the person to alert when
  a problem is detected in the processed data
  ========================================================================
*/
const int alertNum[] = { 9, 5, 4, 8, 3, 0, 6, 1, 8, 3 };
const int ALERT_NUM_LENGTH  = 10;

/*========================================================================
  Controller Handle

  This is a reference to a ControllerTransferProtocol object. It is
  used to poll the controller and populate the data array
  ========================================================================
*/
ControllerTransferProtocol controller;

/*========================================================================
  ARDUINO CONTROL METHODS

  setup() -- runs once, initializes outputs
  loop() -- main control loop
  ========================================================================
*/

void setup()
{
  initializeController();
  initializeSerialPort();
  setOutputPins();
  setInputPins();
  initializeOutputPinStates();
  initializeInterrupt();

  // debug
  pinMode( LED_PIN, OUTPUT );
  digitalWrite( LED_PIN, LOW );

  // indicate to the user that we're starting up
  outputBlink( 4, 500 );

  // initialize data array
  constructDataFrame( controller );

  togglePower();
  delay( 20000 ); // wait 20 secs for startup
}

void loop()
{
  // check if time to do a transfer
  if( executeDataTransfer )
```

```
  {
    // output our data frame
    startCall();
    outputDataFrame( valArr,         NUM_DIAGNOSTIC_PARAMS ); // current value
    outputDataFrame( valArrMinusOne, NUM_DIAGNOSTIC_PARAMS ); // current value - 1
    outputDataFrame( valArrMinusTwo, NUM_DIAGNOSTIC_PARAMS ); // current value - 2
    terminateCall();
    setExecuteTransfer( false );
    resetLoopCounter();
  }

  // poll for data every 30 seconds
  if( LOOP_COUNTER < THIRTY_SECONDS )
  {
    incrementLoopCounter();
    delay( 300 );
  }
  else if( LOOP_COUNTER >= THIRTY_SECONDS )
  {
    // update the data fields
    resetLoopCounter();
    logIfIntervalElapsed();
    constructDataFrame( controller );
    validateDataFrame();
    outputBlink( 1, 30 );
  }
}

/*=====================================================================
  APPLICATION FUNCTIONS

  alert() -- raise an alert to overseers that there is a problem requiring attention
  clearDataFrame() -- clears out the values in the data array before they are polled again
  constructDataFrame() -- polls the controller and populates the data array to transfer
  incomingCallISR() -- interrupt service routine for incoming calls
  incrementLoopCounter() -- increments the main loop update variable
  initializeController() -- initializes the controller transfer protocol object
  initializeInterrupt() -- initializes external interrupts
  initializeOutputPinStates() -- initializes the state of the output pins
  initializeSerialPort() -- initializes all serial communication.
  logIfIntervalElapsed() -- decides to log the data in the queue if the specified interval is up
  outputBlink() -- debug output outputBlinks indicating a process is starting
  outputField() -- receives an int, outputs as keypresses
  outputDataFrame() -- receives an array of values, outputs to cell phone
  populateDataArray() -- fills the data array with the current values of the diagnostic params
  pushKey() -- pushes a single key
  queueData() -- queues the data in the logger arrays
  resetLoopCounter() -- resets the main loop update variable
  setExecuteTransfer() -- toggles the flag that initiates a data transfer
  setInputPins() -- sets pins as inputs
  setOutputPins() -- sets pins as outputs
  startCall() -- accepts an incoming call
  terminateCall() -- terminates a current call
  togglePower() -- turns power on and off
  validateDataFrame() -- ensures that values in the data frame are within expected limits

  =====================================================================
*/

/*=====================================================================
  alert() -- raise an alert to overseers that there is a problem requiring attention
*/
void alert()
{
  int i;
  // dial alert number
  for( i = 0; i < ALERT_NUM_LENGTH; ++i )
  {
    pushKey( decimalArray[ alertNum[i] ] );
  }
```

```
  // push send, delay 10s for call to initiate
  startCall();
  delay( 10000 );

  // for ~20 seconds output '1' and '#' alternatively as an alarm
  for( i = 0; i < 20; ++i )
  {
    pushKey( ONE );
    pushKey( POUND );
  }

  // end the call
  terminateCall();
}

/*=========================================================================
  clearDataFrame() -- clears out the values in the data array before they are polled again
*/
void clearDataFrame()
{
  panelVoltage = 0;
  panelAmperage = 0;
  batteryVoltage = 0;
  batteryAmperage = 0;
  totalKilowattHours = 0;
  populateDataArray();
}

/*=========================================================================
  constructDataFrame() -- accepts an incoming call
*/
void constructDataFrame( ControllerTransferProtocol ctp )
{
  clearDataFrame();
  panelVoltage = ctp.getPwrSrcVoltage();
  panelAmperage = ctp.getChargeCurrent();
  batteryVoltage = ctp.getBatteryVoltage();
  batteryAmperage = ctp.getLoadCurrent();
  totalKilowattHours = ctp.getTotalKilowattHrs();
  populateDataArray();
}

/*=========================================================================
  incomingCallISR() -- interrupt service routine for incoming calls
*/
void incomingCallISR()
{
  static int mutex = 1;
  if( mutex == 1 )
  {
    noInterrupts();              // disable interrupts for critical section
    --mutex;                     // toggle the mutex -- stops other execution
    setExecuteTransfer( true );  // enable data transfer from main control loop
    ++mutex;                     // toggle the mutex -- releases control
    interrupts();
  }
}

/*=========================================================================
  incrementLoopCounter()  -- increments the main loop update variable
*/
void incrementLoopCounter()
{
  ++LOOP_COUNTER;
}

/*=========================================================================
  initializeController() -- initializes the controller transfer protocol object
*/
void initializeController()
{
```

```
   // initialize the controller
   controller = ControllerTransferProtocol();
}

/*========================================================================
  initializeInterrupts() -- initializes external interrupts
*/
void initializeInterrupt()
{
  // external interrupt 0 on pin 2
  attachInterrupt( interruptZero, incomingCallISR, CHANGE );
}

/*========================================================================
  initializeOutputPinStates() -- initializes the state of the output pins
*/
void initializeOutputPinStates()
{
  digitalWrite( NODE_1, LOW );
  digitalWrite( NODE_2, LOW );
  digitalWrite( NODE_3, LOW );
  digitalWrite( NODE_A, LOW );
  digitalWrite( NODE_B, LOW );
  digitalWrite( NODE_C, LOW );
  digitalWrite( NODE_D, LOW );
  digitalWrite( ACCEPT, LOW );
  digitalWrite( PWR, LOW );
}

/*========================================================================
  incomingCallISR() -- interrupt service routine for incoming calls
*/
void initializeSerialPort()
{
  // initialize serial baud rate
  Serial.begin(9600);
}

/*========================================================================
  logIfIntervalElapsed() -- decides to log the data in the queue if the
  specified interval is up
*/
void logIfIntervalElapsed()
{
  // only initialized on first call
  static int interval = 0;
  if( ++interval >= LOG_INTERVAL )
  {
    interval = 0;
    queueData();
  }
}

/*========================================================================
  outputBlink() -- debug output outputBlinks indicating a process is starting
*/
void outputBlink( int beats, int delay_period )
{
  for(int i = 0; i < beats; ++i ){
    digitalWrite( LED_PIN, HIGH );    // set the LED on
    delay( delay_period );            // wait
    digitalWrite( LED_PIN, LOW );     // set the LED off
    delay( delay_period );
  }
}

/*========================================================================
  outputDataFrame() -- receives an array of values, outputs to cell phone
*/
void outputDataFrame( double* valuesArray, int numVals )
{
```

```
  // output the array
  for(int i = 0; i < numVals; ++i )
  {
    outputBlink( 1, 30 );       // outputBlink to indicate separation
    pushKey( POUND );                   // indicate field separator
    outputField( valuesArray[i] );     // test values!!
  }
  outputBlink( 1, 30 );         // outputBlink to indicate separation
  pushKey( POUND );
}

/*=========================================================================
  outputField() -- receives an int, outputs as keypresses
*/
void outputField( double num )
{
  // takes a positive number, outputs it to the keypad
  if( num > 0.001 ) // ensure positive & sufficiently large value
  {
    // first, we convert from double to long, preserving
    // TWO decimal places by multiplying 10^2.
    unsigned long val = long( num * 100 );

    // figure out the size of num
    unsigned long temp = val;
    int digitCount = 0;
    while ( temp > 0 )
    {
      digitCount += 1;
      temp = temp / 10; // chop off one digit at a time
    }

    // break the number into an array of digits
    if( digitCount > 0 )
    {
      int digit;
      int i = digitCount - 1 ; // start at the end of the array
      int buffer[ digitCount ];
      while( val > 0 )
      {
        digit = val % 10; // grab the one's place
        buffer[ i-- ] = digit; // store the digit in the array
        val = val / 10; // chop off the one's place.
      }

      // output the value to the keypad
      for( i = 0; i < digitCount; ++i )
      {
        int decimalVal = buffer[ i ];
        pushKey( decimalArray[ decimalVal ] );
      }
    }
  }
  else { // equivalent of zero
    pushKey( decimalArray[ 0 ] );
  }
}

/*=========================================================================
  populateDataArray() -- fills the data array with the diagnostic params
*/
void populateDataArray()
{
  // populate data array
  valArr[0] = panelVoltage;
  valArr[1] = panelAmperage;
  valArr[2] = batteryVoltage;
  valArr[3] = batteryAmperage;
  valArr[4] = totalKilowattHours;
}
```

```
/*=========================================================================
  pushKey() -- pushes a single key
*/
void pushKey( const int* key )
{
  digitalWrite( key[0], HIGH );   // assert pin 1 HIGH
  digitalWrite( key[1], HIGH );   // assert pin 2 HIGH
  delay( 300 );                   // time required for transistor delay / pin debouncing
  digitalWrite( key[0], LOW );    // reassert pin 1 LOW
  digitalWrite( key[1], LOW );    // reassert pin 2 LOW
  delay( 300 );                   // time required for transistor delay / pin debouncing
}

/*=========================================================================
  queueData() -- queues the data in the logger arrays
*/
void queueData()
{
  int i;
  // move data in the n-1 position into n-2 position
  for( i = 0; i < NUM_DIAGNOSTIC_PARAMS; ++i )
  {
    valArrMinusTwo[i] = valArrMinusOne[i];
  }
  // move data in the nth position into the n-1 position
  for( i = 0; i < NUM_DIAGNOSTIC_PARAMS; ++i )
  {
    valArrMinusOne[i] = valArr[i];
  }
}

/*=========================================================================
  resetLoopCounter() -- resets the main loop update variable
*/
void resetLoopCounter()
{
  LOOP_COUNTER = 0;
}

/*=========================================================================
  setExecuteTransfer() -- toggles the flag that initiates a data transfer
*/
void setExecuteTransfer( boolean value )
{
  executeDataTransfer = value;
}

/*=========================================================================
  setInputPins() -- sets pins as inputs
*/
void setInputPins()
{
  // assert pins as inputs
  pinMode( INCOMING_CALL, INPUT );
}

/*=========================================================================
  setOutputPins() -- sets pins as outputs
*/
void setOutputPins()
{
  // assert pins as outpus
  pinMode( NODE_1, OUTPUT );
  pinMode( NODE_2, OUTPUT );
  pinMode( NODE_3, OUTPUT );
  pinMode( NODE_A, OUTPUT );
  pinMode( NODE_B, OUTPUT );
  pinMode( NODE_C, OUTPUT );
  pinMode( NODE_D, OUTPUT );
  pinMode( ACCEPT, OUTPUT );
  pinMode( PWR, OUTPUT );
```

```
}

/*=========================================================================
  startCall() -- accepts an incoming call
*/
void startCall()
{
  pushKey( START_CALL );        // accept the call (assumes interrupt fired)
  delay( 1000 );                 // wait 1 second for call to begin
}

/*=========================================================================
  terminateCall() -- terminates a current call
*/
void terminateCall()
{
  pushKey( POWER ); // hit the power button to end a call
}

/*=========================================================================
  togglePower() -- turns power on and off
*/
void togglePower()
{
  digitalWrite( POWER[0], HIGH );
  delay( 4000 ); // delay 4 seconds
  digitalWrite( POWER[0], LOW );
}

/*=========================================================================
  validateDataFrame() -- ensures that values in the data frame are within expected limits
*/
void validateDataFrame()
{
  //valArr[0] = Panel Voltage
  //valArr[1] = Panel Amperage
  //valArr[2] = Battery Voltage
  //valArr[3] = Battery Amperage
  //valArr[4] = Total Kilowatt Hours
  double currBatteryVoltage = valArr[2];

  // TODO, make a better abstraction of this comparisons
  if( currBatteryVoltage <= LOW_VOLTAGE_CUTOFF )
  {
    // an error has been detected. alert the masses!
    alert();
  }
}
```