Theme Report 2: Reason

COMEPNG 2DX3: Microprocessor Systems Project

Thursday, March 13th, 2025

Joey McIntyre (400520473)

Theme

In an intelligent system, reasoning is the ability of the system to handle observed inputs, evaluate data, and make reasoned decisions based on established rules or acquired patterns. While observation centers on collecting the raw data from the surroundings, the reasoning theme allows the system to analyze that information and decide on a suitable response. This theme is crucial when converting raw inputs into significant actions and enabling embedded systems to operate independently and effectively in practical situations.

In labs 4-6 of the course, reasoning is demonstrated through activities like handling inputs from a keypad, interpreting binary values, and combining various components to make sound decisions. By utilizing algorithms like state machines, lookup tables, and logical condition evaluations, intelligent systems can analyze a user's inputs and system conditions to produce the suitable response. The reason theme is crucial when ensuring that embedded systems not only passively gather data but also make choices that direct their actions, which connects to the final act theme discussed in subsequent labs.

Background

The reason theme is essential in intelligent systems as it enables a system to analyze observed data, perform logical operations, and make decisions prior to taking action. The observe theme examined earlier focuses on gathering input data from sensors and user interfaces, while reasoning guarantees that the data is processed meaningfully, enabling the system to react correctly. Without reasoning capabilities, an intelligent system simply cannot handle complex inputs, identify patterns, or change its behavior based on current conditions.

In embedded systems, reasoning is implemented using finite state machines, decision trees, lookup tables, and algorithm processing to assess input conditions and identify the suitable outputs. In Lab 4 (Duty Cycle and Pulse Timing), logic was applied to adjust pulse width modulation signals to control an LED and a stepper motor. The system needed to evaluate the duty cycle input and modify the output signal where necessary, illustrating an essential type of reasoning where the microcontroller understands input values and alters hardware performance. In lab 5 (Peripheral Interfacing), logic was applied when scanning and interpreting the 4x4 keypad. The system needed to identify the key that was pressed and associate it with a matching binary value. This process involved identifying the signal from each distinct key press, correlating it with a preset lookup table, and delivering the accurate result to the systems memory and the microcontrollers LED display. Finally, in Lab 6 (Deliverable 1 for the project), reasoning was critical for essentially integrating all the peripherals we have used in labs up to this point. This includes the stepper motor, push buttons (both internal and external), and the on-board LEDs, using a state machine approach. The microcontroller was required to evaluate the user input from any of the four buttons, set the motor status, and adjust the LEDs appropriately to show correct operation, demonstrating a sophisticated decision-making process.

These reasoning skills are crucial for intelligent systems to function successfully in real-world scenarios. Whether we are talking about robotics, automation, or smart devices, the capacity to analyze inputs and make rational decisions is what sets an intelligent system apart from a merely reactive one. By learning to apply reasoning in embedded systems, we acquire the capability to create smarter, more autonomous solutions that effectively adapt to constantly evolving environments.

Theme Exemplars

To demonstrate the reason theme in intelligent systems, I will present two examples from our lab work that show how the system interprets the data it observed, processes logical decisions, and determines appropriate responses. These examples come from Lab 5 (Peripheral Interfacing: Keypad Decoding) and Lab 6 (Deliverable 1 for the project).

The first example comes from Lab 5, Milestones 7.2 and 7.3, which was the keypad decoding and interpretation. In this lab, I implemented the 4x4 keypad interface to detect key presses, identify the corresponding binary values, and then output them to an LED display. The reasoning aspect comes into play during key scanning and decoding, where the system must identify which key was pressed, match it to a stored lookup table, and output the corresponding value.

The first step is the key scanning process. The keypad is arranged in a 4-row by 4-column matrix, where each row is connected to Port E (PE3:0) and each column is connected to Port M (PM3:0). To detect a key press, the microcontroller activates one row at a time by setting it HIGH while keeping all the other rows LOW. It then checks the column inputs to see if any column reads HIGH, which indicates a pressed key. Figure 1 gives a visual representation of the internal wiring matrix of the 16-button keypad, showing how each button press connects to a specific row and column, which enables the microcontroller to detect key presses using reasoning (a scanning technique). [1]

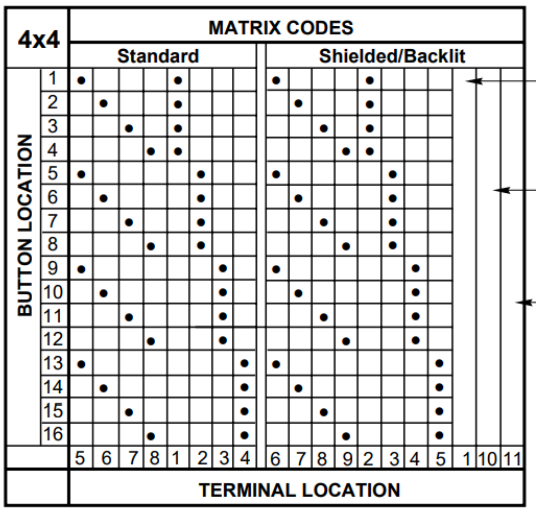

Figure 1: 4x4 Keypad Matrix Configuration and Terminal Mapping

The next step is the key decoding, which is the heart of the reasoning process. Each key press corresponds to a unique binary scanning input code. A lookup table, which is stored in memory, maps these scan codes to actual binary values. This decoded binary value is stored in Keil memory for further use. The final step is outputting the binary value to the microcontrollers on-board LEDs. The decoded key value is sent to four GPIO output pins, where each bit controls an LED. If a key is pressed, the LEDs display the correct binary representation for each pressed key. Using the Keil debugger, the variable watch function shows that the lookup table correctly converts the scanning input code into the expected binary output by monitoring the decoded key variable, as well as the relevant Port M and Port E value.

Key Pressed	Binary Output
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
*	1110
#	1111

Figure 2: 4x4 Keypad Binary Output Table

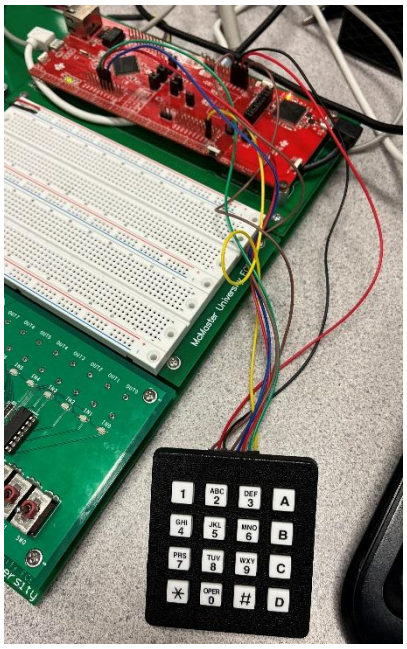

Figure 3: Keypad Interface with LED Output

Figure 2 shows the binary output mapped to each key press on the 4x4 keypad. Figure 3 shows the physical implementation of the keypad connected to the microcontroller, outputting the correct binary value on the on-board LEDs (current button pressed is 4 – correlating to a binary output of 0100). Figure 4 shows the relevant code for scanning the pressed key and decoding the key to output the correct binary value to the LEDs.

```
uint8_t Read_Keypad(void)
{
  uint8_t row, col;
  uint8_t key_value = 0xFF;

  for (row = 0; row < 4; row++)
  {
    GPIO_PORTE_DATA_R = ~(1 << row);
    for (col = 0; col < 4; col++)
    {
      if ((GPIO_PORTM_DATA_R & (1 << col)) == 0)
      {
        key_value = ((GPIO_PORTM_DATA_R & 0x0F) << 4) | (GPIO_PORTE_DATA_R & 0x0F);
        return key_value;
      }
    }
  }
  return key_value;
}
```

```c
uint8_t Decode_Key(uint8_t key_value)
{
  switch (key_value)
  {
    case 0xEE: return 0x1;  //button 1
    case 0xDE: return 0x2;  //button 2
    case 0xBE: return 0x3;  //button 3
    case 0x7E: return 0xA;  //button A
    case 0xED: return 0x4;  //button 4
    case 0xDD: return 0x5;  //button 5
    case 0xBD: return 0x6;  //button 6
    case 0x7D: return 0xB;  //button B
    case 0xEB: return 0x7;  //button 7
    case 0xDB: return 0x8;  //button 8
    case 0xBB: return 0x9;  //button 9
    case 0x7B: return 0xC;  //button C
    case 0xE7: return 0xE;  //button *
    case 0xD7: return 0x0;  //button 0
    case 0xB7: return 0xF;  //button #
    case 0x77: return 0xD;  //button D
    default: return 0xFF;   //default
  }
}
void OutputToLEDs(uint8_t value)
{
  GPIO_PORTN_DATA_R = ((value & 0x01) << 1) | ((value & 0x02) >> 1);
  GPIO_PORTF_DATA_R = ((value & 0x04) << 2) | ((value & 0x08) >> 3);
}
int main(void)
{
  PortE_Init();
  PortM_Init();
  PortN_Init();
  PortF_Init();

  while (1)
  {
    key_value = Read_Keypad();
    if (key_value != 0xFF)
    {
      last_key = key_value;
      decoded_key = Decode_Key(key_value);
      OutputToLEDs(decoded_key);
    }
  }
}
```

Figure 4: Relevant C Code for Key Scanning, Decoding, and Output

The second example comes from Lab 6, which acts as the initial project deliverable. This lab centered on gathering inputs from both the onboard and external buttons to control the function of a stepper motor. The system had to identify button presses, determine the appropriate motor state, and update the on-board indicator LEDs as necessary.

The first step is monitoring button inputs and state handling. Four buttons were used for motor control: Button 0 (on-board) toggled the motor ON or OFF, Button 1 (on-board) changed the motor direction between clockwise and counterclockwise, Button 2 (external pushbutton) toggled the rotation step size between 45 degrees and 11.25 degrees, Button 3 (external pushbutton) returned the motor to its home position. The GPIO inputs for each button were continuously monitored and a finite state machine was implemented to track the motors behaviour.

The next step is the motor control logic which demonstrates the reasoning theme. If Button 0 is pressed, the FSM toggles the motor state (OF/OFF) and updates LED 0 (ON indicates the motor is ON). If Button 1 is pressed, the motor reverses its direction, and LED 1 is updated accordingly (ON indicates clockwise rotation). If Button 2 is pressed, the step size is toggled between 45 and 11.25 degrees, and LED 2 is updated (ON when step size is 11.25 degrees). LED 3 blinks in real-time to show each step of the rotation. This provides valuable debugging feedback as it shows

whether button 2 is functioning correctly. If Button 3 is pressed, the motor returns to the home position of 0 degrees, and all LEDs are turned off.

The last stage is confirming results. The FSM effectively monitored and modified the motor's state according to the information received from the buttons. The motor reacted accurately to every button press, confirming that the system's reasoning is functioning properly. The Keil debugger was utilized to monitor GPIO registers to verify that the input detection and motor state updates were accurate. The on-board LEDs always indicated the motor’s current state, further confirming that the system is processing the data as intended.

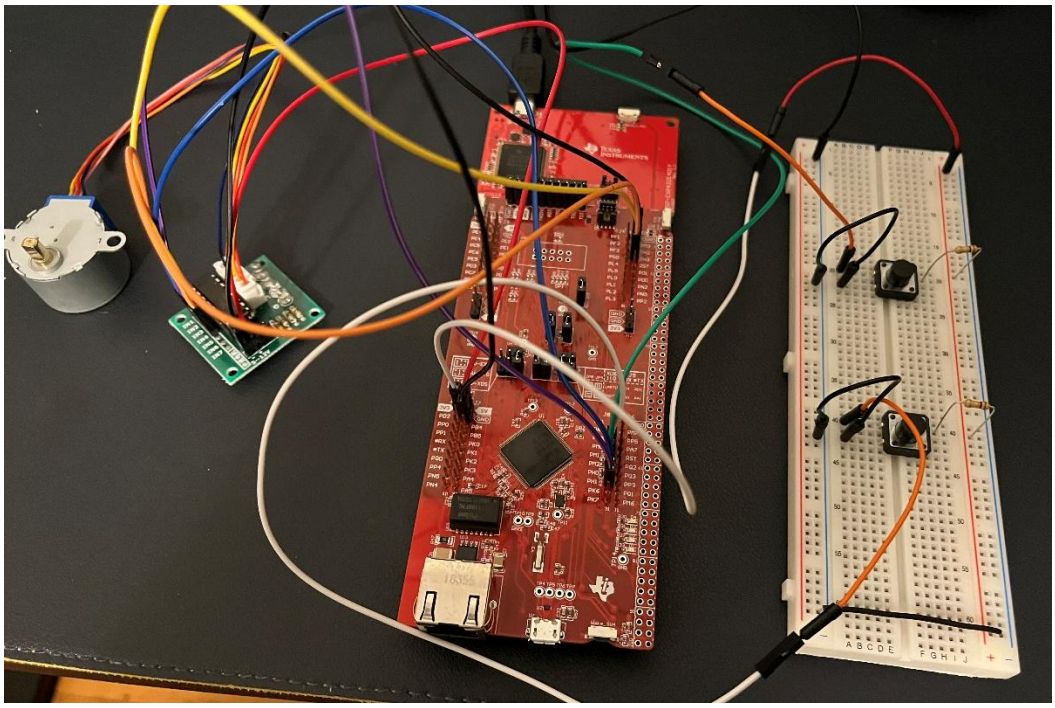

Figure 5: Stepper Motor Setup with On-Board LEDs Indicating Current State

```
//main function
int main(void) {
    //initialize all the ports and system clock
    PortH_Init();
    PortJ_Init();
    PortM_Init();
    PortN_Init();
    PortF_Init();
    SysTick_Init();

    uint8_t motorOn = 0;                        //motor status (0 = off, 1 = On)
    uint8_t direction = 0;                      //motor direction (0 = clockwise, 1 = counterclockwise)
    uint8_t lastPJ = 0x03;                      //previous state of PJ buttons
    uint8_t lastPM = 0x03;                      //previous state of PM buttons

    while (1) {
        uint8_t pj = GPIO_PORTJ_DATA_R & 0x03;  //read the onboard (PJ0 and PJ1) buttons
        uint8_t pm = GPIO_PORTM_DATA_R & 0x03;  //read external (PM0 and PM1) buttons

        //button 0: start / stop motor - PJ0
        if ((lastPJ & 0x01) && !(pj & 0x01)) {  //if button is pressed - checks for for a falling edge because button is active low (pull up resistor)
            motorOn ^= 1;
            GPIO_PORTN_DATA_R = (motorOn) ? (GPIO_PORTN_DATA_R | (1 << 1)) : (GPIO_PORTN_DATA_R & ~(1 << 1));
        }

        //button 1: toggle direction - PJ1
        if ((lastPJ & 0x02) && !(pj & 0x02)) {
            direction ^= 1;
            GPIO_PORTN_DATA_R = (direction == 0) ? (GPIO_PORTN_DATA_R | (1 << 0)) : (GPIO_PORTN_DATA_R & ~(1 << 0));
        }

        //button 2: toggle step angle - PM1
        if ((lastPM & 0x02) && !(pm & 0x02)) {
            angleMode ^= 1;
            GPIO_PORTF_DATA_R = (angleMode == 1) ? (GPIO_PORTF_DATA_R | (1 << 4)) : (GPIO_PORTF_DATA_R & ~(1 << 4));
        }

        //button 3: return home - PM0
        if ((lastPM & 0x01) && !(pm & 0x01)) {
            motorOn = 0;
            GPIO_PORTN_DATA_R &= ~(1 << 1);
            goHome();
        }

        //run the motor if its enabled
```

```
        if (motorOn) {
            spinStep(direction);
        } else {
            GPIO_PORTH_DATA_R = 0x00;          //stop motor immediately
            GPIO_PORTN_DATA_R &= ~(1 << 1);    //turn OFF LED0
            GPIO_PORTN_DATA_R &= ~(1 << 0);    //turn OFF LED1
            GPIO_PORTF_DATA_R &= ~(1 << 4);    //turn OFF LED2
            GPIO_PORTF_DATA_R &= ~(1 << 0);    //turn OFF LED3
        }

        lastPJ = pj;
        lastPM = pm;
    }
}

//spin motor one step
void spinStep(uint8_t direction) {
    if (absoluteposition >= 2048){
      absoluteposition = 0;
    } else{absoluteposition += 4;}

    if (angleMode == 1){                        //small steps - 11.25 degrees - LED3 blinks every 64 steps
      if (absoluteposition % 64 == 0){
      GPIO_PORTF_DATA_R |= (1 << 0);            //blink LED3 each step
      SysTick_Wait(1200000);
      GPIO_PORTF_DATA_R &= ~(1 << 0);
    }

    }
    else if (angleMode == 0){                   //large steps - 45 degrees - LED3 blinks every 256 steps
      if (absoluteposition % 256 == 0){
      GPIO_PORTF_DATA_R |= (1 << 0);            //blink LED3 each step
      SysTick_Wait(1200000);
      GPIO_PORTF_DATA_R &= ~(1 << 0);
    }
    }


    stepPosition = (direction == 0) ? (stepPosition + 4) % 2048 : (stepPosition == 0) ? 2044 : stepPosition - 4;

    static const uint8_t fullStepSeq[4] = {0x03, 0x06, 0x0C, 0x09};

    if (direction == 0) {                       //clockwise
        for (int i = 0; i < 4; i++) {
            GPIO_PORTN_DATA_R |= (1 << 0);      //turn on LED1
            GPIO_PORTH_DATA_R = fullStepSeq[i];
            SysTick_Wait(1200000 * 0.17);       //for future deliverable, bus speed depends on student number - this number needs to be change according
        }
    } else {                                    //counterclockwise
        for (int i = 3; i >= 0; i--) {
            GPIO_PORTH_DATA_R = fullStepSeq[i];
            SysTick_Wait(1200000 * 0.17);
        }
    }
}

//return home function
void goHome(void) {
    if (stepPosition == 0) return;  //already at home

    uint16_t cw_steps = (2048 - stepPosition) % 2048;
    uint16_t ccw_steps = stepPosition;
    uint8_t dir = (cw_steps <= ccw_steps) ? 0 : 1;
    uint16_t stepsToMove = (dir == 0) ? cw_steps : ccw_steps;
    stepsToMove /= 4;

    for (uint16_t i = 0; i < stepsToMove; i++) {
        spinStep(dir);
    }
}
```

Figure 6: Relevant Code for Motor & LED Control

Figure 5 shows the physical implementation of the stepper motor, and the pushbuttons connected to the microcontroller (current state is off). Figure 6 shows all the relevant code for how the motor is controlled by the button inputs, and how the current motor state is visualised by the onboard LEDs.

Debugging Exemplar

The primary method of debugging in labs 4-6 was using the built in Keil debugger to monitor register values. However, since I used this technique as my debugging exemplar for Theme Report 1 (Observe), I need to discuss a time I used a different strategy. In Lab 6 (first project deliverable), I had a issue where Button 3, which is supposed to have the motor return to home, was not functioning correctly. To debug this problem, I used a LED-based approach to determine whether the issue was with the button itself (hardware) or with the code logic (software).

I detected the issue when I was testing the functionality of each button and noticed Button 3 had not effect on the motor's behaviour. This was confusing as all the other buttons were working as expected. I opted to use an LED connected to the Button 3 (pushbutton) to test if the press of the

button is being registered. If the LED illuminated when the button was pressed, it would suggest the problem was related to the software; however, if it remained OFF, it would point to a hardware problem. After installing the LED, I clicked Button 3 multiple times, but the LED didn't illuminate. This indicates that the microcontroller was not getting an input signal from the button. To troubleshoot further, I examined the physical button connections with a multimeter's beep test and discovered that button 3 wasn't completing the circuit when pressed. This indicates that the button I was using was defective; fortunately, I had two identical buttons in my supply kit, so I swapped it out and repeated the LED test. This time, the LED switched properly, and the stepper motor returned to the home position when pressed. This showed that the new button functioned properly, and was further confirmed by checking the GPIO register values in the Keil debugger. This example illustrates the importance of having various debugging techniques, since different issues require distinct approaches.

Synthesis

The reason theme in intelligent systems is demonstrated by the logical processing and decision making that occurred in the previously mentioned exemplars (keypad decoding in Lab 5 and stepper motor control in Lab 6). These two examples involved using different types of input and output, but they both show a system processing raw input data and turning it into meaningful outputs using structured logic.

The keypad decoding example in Lab 5 shows how the microcontroller uses reasoning such as input scanning, interpreting data, and output mapping. The microcontroller receives the raw input signal shown in Figure 1, processes it, and outputs the correct binary value. This shows the structured decision-making process which ensures that every press of a key results in its correct binary representation as shown in Figure 2. The ability to map input signals to their predefined output is a crucial element of reasoning in embedded systems.

Similarly, the stepper motor control system in Lab 6 demonstrates the reasoning process once again. This time, finite state machines and logical conditions are used to determine how the motor should behave depending on the press of each button. Unlike the keypad example where the reasoning is in the conversion of data, this scenario demonstrated real-time reasoning from the user inputs. The microcontroller constantly tracks the state of the motor, updating the LEDs to reflect this, and process when the user presses a button to modify the motors behaviour accordingly.

Together, these two exemplars highlight how reasoning isn’t just about processing static data as this is unrealistic in the real world. It's about allowing the system to obtain external inputs, evaluate existing conditions, and execute the intended action instantly. This concept is crucial for intelligent systems since it ensures that embedded systems are capable of performing beyond simple responses. They can make systematic choices based on logic and set criteria, which is essential for all practical applications.

Reflection

Through these exemplars, I gained a deeper understanding of how reasoning is used and implemented by intelligent systems. The keypad decoding example strengthened by understanding of how microcontrollers interpret confusing input signals like that of a 4x4 keypad, and the techniques this involves (scanning, lookup tables, binary mapping). It really reinforced the importance of an embedded systems ability to process data in a organized way.

The stepper motor control example from the project deliverable provided me with an understanding of how systems utilize finite state machines to make real-time decisions. This system was unique because it required the system to process user inputs in real time and correctly change the state of the motor and LEDs. This made me appreciate how reasoning allows intelligent systems to function dynamically and adapt to real-world interactions.

Furthermore, the debugging example where I used a LED to identify a faulty button in Lab 6 proved the significance of understanding multiple different debugging methods. Software debugging tools like the Keil debugger are effective in many situations, but this hands-on experience demonstrated that without physical troubleshooting, hardware problems can frequently be mistaken for software issues. This process how diverse debugging strategies when designing embedded systems can be very beneficial.

Overall, this theme report improved my understanding of how microprocessors process inputs logically to generate meaningful outputs. It also showed me the importance of reasoning when designing autonomous and intelligent embedded systems, and how these experiences have taught me how to take raw sensor inputs and turn them into real-world actions. Looking ahead, this understanding will be crucial when working on the second deliverable for the final course project.

References

[1] Grayhill, “Standard Keypads,” 96BB2-006-F datasheet, https://www.micro-semiconductor.com/datasheet/a4-96AB2-152-F.pdf