ELEC 327

Xin Huang
Jonathan Nguyen
Michael Tsehaie

# ELEC 327 MSP430G2553 IR Motor Control

**System Concept and Design:**

The concept of our project is simple in theory but a bit more complicated in practice. We would be exploring the use of motors and infrared signal detection with an MSP430; concepts that we have not really covered extensively. We had planned on having these 3 stages of development for the project:

1. Vehicle motors driven by MSP430 after reading signals from an IR remote
2. Remote activating a self-roaming mode with vehicle utilizing ultrasonic range finder to avoid obstacles and reroute
3. Creating our own IR remote with a potentiometer as wheel; LEDS and speakers to pimp out the vehicle

Our minimal viable product would be development stage #1, as the requirements for this project were to incorporate motion control. Stage 2 of development would have been a cool feature to have that adds extra value. Stage 3 would just be for uniqueness and pride.

**System Architecture:**

Need some more modification here and above

1. IR remote send commands into the environment
2. IR signal hopefully gets picked up by IR detector
3. MSP430 gets woken up from LPM0 to read and analyze the rising and falling edges of signal
4. Binary signal gets converted to HEX
5. HEX signal checked across a table for corresponding motor control inputs
6. Motor control inputs are used to enable motors and set speed via PWM
7. Motor driver drives the motor based on these inputs
8. Success

**IR Detection:**

Materials:

1. 21 Button NEC IR remote: http://www.adafruit.com/products/389
2. IR Sensor TSOP38238: http://www.digikey.com/product-detail/en/TSOP38238/751-1227-ND/1681362

The majority of infrared (IR) controllers transmit a digital code using a 38 kHz IR signal. There are some that transmit at 40 kHz, but the 21 button remote we were using transmitted at 38 kHz using an NEC code output with a 940nm IR LED.

We used an IR receiver that came in a 3-pin package. By simply connecting VCC (+2.5V to +5.5V) to pin 3 and GND to pin 2, we were able to get a signal out on pin 1. The IR Receiver, we were using was the TSOP38238 (Datasheet) but any IR receiver will work as long as it is able to detect IR signals at the 940nm wavelength.
When any button is pressed the data sequence begins as follows:

1. A low pulse ~9.2ms wide starts the sequence.
2. The first data pulse follows ~4.4ms later.
3. Data pulses are low pulses ~0.74ms wide.

The spacing before the next data pulse determines whether the bit is zero or one.

- For a "zero," the spacing to the next pulse is ~1.2ms
- For a "one," the spacing to the next pulse is ~2.4ms

The total number of bits transmitted is either 32 or 64 bits; but in our case it is 32 bits.
There are four ways of interpreting the bits received:

1. We can consider short time spacing between clocks (1.2ms) to be "zero" and long intervals (2.4ms) to be "one".
2. Or we can invert the logic of each bit with 1.2ms to be a “one” and 2.4ms to be “zero”
3. We can choose if the first bit received is the least significant bit (LSB) of the data stream.
4. Or we can choose the first bits to be the most significant bit (MSB) of the data stream.

The difficult part was to come up with a mechanism in order to capture and decode the received data stream. I decided to use the MSP430G2553’s timer modules to be able to precisely record and decode the IR signals. The timer module has some great functionality that we will exploit:

- Input capture - This feature allows us to take a time stamp of when an external event occurs. The trigger can be a rising edge, a falling edge or both rising and falling edges of an input signal.
- Output compare - We can set a timer register to a specific count and an interrupt is generated when the timer register reaches this preset count.

Because I am using an IR remote, the input capture mode would be what I want as I want to be able to capture the rising and falling edges of the IR signal. The input capture will be able to determine the time between successive falling edges and from that I will be able to decode the “zeros” and “ones” in the data stream. But because I want to be able to know when the end of one data stream ends and when another one begins, I will also have to use the output compare. The output compare will create a timeout when there are no falling edges appearing within

~120ms, so I know that a signal has completed and I can stop reading and do not accidently combine 2 IR signals together.

The MSP430G2553 has two timer modules, TIMER0 and TIMER1. I am going to use 2 interrupt vectors assigned to TIMER0 for reading and decoding IR signals. TIMER0_A0_VECTOR is used when the Capture/Compare interrupt flag CCIFG in register TA0CCTL0 is set. TIMER0_A1_VECTOR is used for all other interrupts, including CCIFG in TA0CCTL1, CCIFG in TA0CCTL2 and TAIFG in TA0CTL.
In this case, TA0IV is used as an efficient interrupt priority encoder in order to process multiple interrupts. If TA0IV = 0x02, capture or compare associated with register TA0CCR1 occurred and interrupt flag CCIFG in TA0CCTL1 is set. If TA0IV = 0x04, capture or compare associated with register TA0CCR2 occurred and interrupt flag CCIFG in TA0CCTL2 is set. If TA0IV = 0x0A, timer register TA0R overflow occurred and interrupt flag TAIFG in TA0CTL is set. When servicing TIMER0_A1 interrupts, the TA0IV register must be read in order to clear the interrupt request flags.

TIMER0 Input Capture0 is assigned to port P1.1 (pin-3). We connect the output of the IR Sensor Module to this pin. We initialize the hardware to use P1.1 as Input Capture whenever a falling edge is detected. We use TIMER0_A1_VECTOR to process a timeout condition to tell us that the data transmission has ended. Since we are using P1.1 as the input pin, we have already assigned TA0CCR0 to be the input capture register, so we cannot use that the default timeout counter. Hence we have to use the timer in the CONTINUOUS COUNT mode and will stop the timer when the timeout is reached.

The basic MCU internal clock is 1MHz. The timer is configured with a divide-by-8 prescaler, such that each timer count represents 8μs.
LED1 at P1.0 (pin-2) is used as an indicator when data is being received. LED1 goes off when the timeout has occurred.
The TIMEOUT has been set for about 120ms to avoid repeated action when the key is held down continuously.

Basically this IR module and the code should be able to read in IR signals sent in any format, not just NEC, since I am reading in the binary bits. I decided to condense the 32 bit binary into a length 8 HEX code. So given my 21 button remote I was able to decode my IR signals into the following HEX command table:

| **REMOTE COMMAND** | **HEX VALUE** |
|---|---|
| VOL_DOWN | FF00DF00 |
| PLAY | F708DF00 |
| VOL_UP | FB04DF00 |
| SETUP | FD02DF00 |
| UP | F50ADF00 |

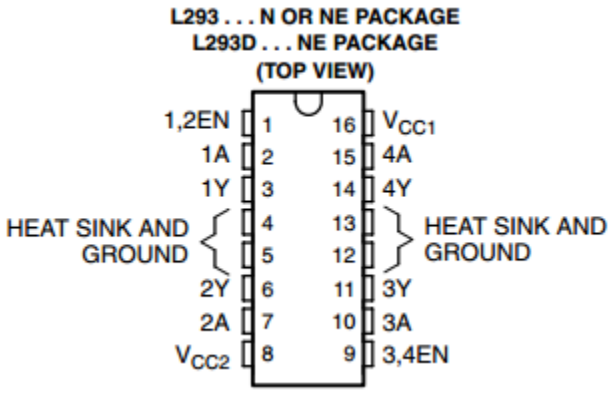
| | |
|---|---|
| STOP | F90GDF00 |
| LEFT | FE01DF00 |
| ENTER | F609DF00 |
| RIGHT | FA05DF00 |
| PLUS | FC03DF00 |
| DOWN | F40BDF00 |
| BACK | F807DF00 |
| | |
| ONE | 7F80DF00 |
| TWO | 7788DF00 |
| THREE | 7B84DF00 |
| FOUR | 7D82DF00 |
| FIVE | 758ADF00 |
| SIX | 7986DF00 |
| SEVEN | 7E81DF00 |
| EIGHT | 7689DF00 |
| NINE | 7A85DF00 |

**Driving the Motor:**

Materials:

1. Motors from our vehicle
2. Motor Driver L293D: http://www.adafruit.com/datasheets/l293d.pdf

The motor driver we are using is a quadruple high-current half-H drivers designed to provide bidirectional drive currents up to 600 mA between 4.5V and 36V. The inputs are TTL compatible. TTL, short for transistor-transistor logic, is a class of digital circuits that uses transistors to perform logic gating and amplifying functions. Below is the pinout of the motor driver.




Pin 1 and Pin 9 are used to enable the motor 1 and motor 2, respectively. When an enable pin is high, their outputs pins 3 & 6, and pins 11 & 14, are active and in phase with their inputs pins

2&7, and pins 10&15, respectively. When the enable inputs are low, the outputs are off. The circuit we aim to build using the motor driver is shown below.

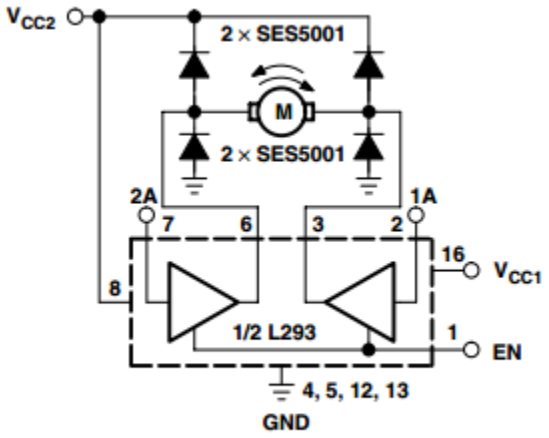



**Figure 5. Bidirectional DC Motor Control**

Unfortunately, the SES5001 diodes were discontinued so we used the MUR220 diode instead. The schematic above allows current to flow through the motor in both directions, depending on pin 2 and pin 7 (and pin 10 and 15 which are hidden). The diodes are used to keep dangerous current from flowing back to the motor driver when the motor suddenly stops or changes direction. The truth table for the schematic is shown below:

| EN | 1A | 2A | FUNCTION |
|---|---|---|---|
| H | L | H | Turn right |
| H | H | L | Turn left |
| H | L | L | Fast motor stop |
| H | H | H | Fast motor stop |
| L | X | X | Fast motor stop |

L = low, H = high, X = don't care

The two motors that move the excavator are configured so they are placed parallel to each other. Therefore, to move forward and backwards, you would have to run both motors at the same time in the same direction. If you wanted to turn left or right, however, you would have to let one motor run in one direction and let the other one run in the other direction. This can be summarized in a table as follows:

| | Motor 1 | | | Motor 2 | | |
|---|---|---|---|---|---|---|
| **EN (1,2)** | **1A** | **2A** | **EN (3,4)** | **3A** | **4A** | **Function** |
| H | L | H | H | L | H | **Move Forward** |
| H | H | L | H | H | L | **Move Backwards** |
| H | L | H | H | H | L | **Move Left** |
| H | H | L | H | L | H | **Move Right** |
| L | X | X | L | X | X | **Stop** |

Using a few case statements, we implemented the above functions depending on the input from the IR receiver.

**Challenges:**

The main challenge of this project was more time management than anything else. Even though we were not experts on motors or infrared protocols, we had become expert tinkerers and datasheet readers over the course of the semester. Because of a delayed parts sourcing schedule, our PCB had some errors in it with some parts that were the wrong package, some pins were also not initialized to be used. Our PCB could have been better if we had made each leg of the MSP430 go out to a pin header so that any problems or future additions can be easily added.

We initially had trouble with our motor driver when we implemented one of the functions in the truth table. We thought that because pins 2 and 7 were input pins, they were low by default when we powered the l293d motor driver. However, when we measured the value of those pins, we realized that some voltage was being outputted even though it was an input pin. For the longest time, we thought that the chip was faulty. Only after careful reading of the datasheet and forcing these pins to be low when need be, did we realize that is how the motor chip functions.

Another frustrating issue we had was with the voltage regulator. In the mid-term project, we had to deal with a non-linear voltage regulator that outputted a different voltage than advertised depending on the current output. This time around, we got a voltage regulator with the appropriate voltage output (which can be confirmed with a DMM) but it did not output enough current to drive the motor chip. Unfortunately, we had to do away with the voltage regulator and use the msp430 launchpad to power the motor driver and the msp430, thereby severely limiting the mobility of our excavator. We hope that you teach how voltage regulators function in future classes of ELEC 327 so that other students won't face issues with choosing the right voltage regulator.