Project 3.7 Double Dabble

Purpose

The purpose of the Double Dabble project is to utilize the shift and add 3 (double dabble) algorithm to convert a 12-bit reading from a potentiometer to a 5-nibble packed BCD value, and echo it using ASCII equivalents through the USART interface.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/Tasks.html#DoubleD

Project GitHub: https://github.com/rohan-development/3.7-DoubleDabble

Theory

There are many different *sets* of data. A set refers to a type of data with some number of elements. If it is a closed set, there are a finite number of elements. If it is an open set, there are an infinite number of elements. While elements of a set, can be represented by characters, when written on paper, a computer cannot represent any type of data aside from binary numbers. All stored data in a computer is either a 0, or a 1. The most common way to interpret a string of 0s and 1s is as a binary number; this is not the only interpretation, however. As long as the computer (or sender and receiver) agree on what set is represented by some number of binary digits, the data can represent any set. For example, if it is agreed that 5-bit value represents the lowercase letter, 00000 could be said to represent a, and 11001 could be said to represent z. This idea gives rise to the American Standard Code for Information Interchange (ASCII). ASCII is utilized to represent characters using a value. An ASCII value is a byte long, although it only uses 7 of the 8 bits. There are 128 possible characters, 95 of which are printing characters, and the remaining 33 are control characters. When an ASCII character is sent or received, only the raw value is sent. Values are chosen carefully: capital and lowercase letters simply differ by 1 bit (see Figure 1).

Figure 1. ASCII Table

Character	ASCII (Bin)	ASCII (Dec)	ASCII (Hex)
0	0011 0000	48	0x30
1	0011 0001	49	0x31
2	0011 0010	50	0x32
3	0011 0011	51	0x33
…			
7	0011 0111	55	0x37
8	0011 1000	56	0x38
9	0011 1001	57	0x39
A	0100 0001	65	0x41
B	0100 0010	66	0x42
C	0100 0011	67	0x43
…			
Z	0101 1010	90	0x5A
a	0110 0001	97	0x61
b	0110 0010	98	0x62
c	0110 0011	99	0x63
…			
z	0111 1010	122	0x7A

Procedure

In order to display anything on the Arduino serial monitor, ASCII must be utilized (see Theory section). In a high-level language, when the user makes a call to the serial monitor, the Serial library converts everything to an ASCII value behind the scenes, before sending the ASCII code over USART. The serial, monitor automatically interprets a number to be its ASCII equivalent. For example, if decimal 51 is received, 51 is not displayed; '3' is displayed because its ASCII value is decimal 51. Therefore, in order to send a number to the serial monitor, it must be first converted to a stream of several ASCII characters.

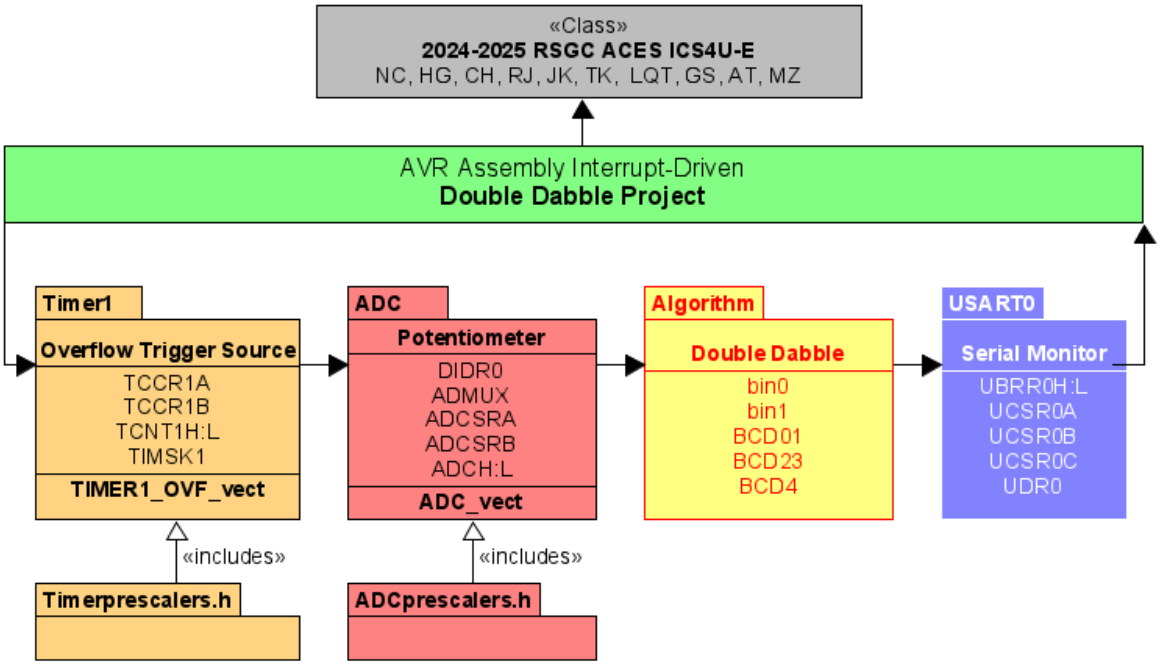


Figure 1. Double Dabble Flowchart

Figure 1 depicts the way the double dabble project prints the value read by the ADC unit on the serial monitor, from a high-level perspective. First, Timer1 is utilized to generate an interrupt upon it overflowing. When the interrupt is generated, the ADC begins a reading, and the Timer 1 overflow vector is called. This ISR simply resets the value in Timer1 to zero. When the ADC reading is completed, the 12-bit value is stored in the ADCL and ADCH register pair and a second interrupt is triggered. This interrupt loads the ADCL and ADCH registers into r18 (bin0) and r19 (bin1), respectively. Next the number that is now split between bin0 and bin1 is separated into its ASCII decimal digits, and sent to the serial monitor via USART.

The bulk of the computation done within this project is the separation into its distinct digits. In a high-level language, the modulo and division operator could be utilized in the same fashion as performed in Project 2.5.3. In assembly language, the modulo operator does not exist. Instead, the binary number must be separated into decimal digits. One candidate to perform this function is binary-coded decimal (BCD) (see Project 1.4:E). BCD essentially represents a number using 4 bits to represent each decimal digit of a number. For example, the binary number 1101 is equivalent to 13 in decimal. As such, its BCD conversion is 0001 0011. By converting the ADC reading to BCD, the reading can be easily converted to a stream of ASCII characters.

In order to convert a standard binary number to BCD, the double dabble, or shift-and-add-3 algorithm is used. This is an extremely simple algorithm that can be used to convert any binary number to a BCD value. The binary number is first set up with appropriate "scratch space" that can hold the converted BCD number (which will be larger). Since each digit will take up a nibble (4 bytes) the scratch space is separated into nibble-sized chunks (see Figure 2). First, the binary input is shifted left, spilling into the BCD scratch space. This is repeated until the value of any 1 nibble is equal to or exceeds the value 5. In this case, 3 is added to that nibble, and the system is shifted again. This runs until the original binary input has been shift *n* times, where n is the bit length of the input.

Figure 2. Double Dabble Sample Conversion

BCD2	BCD1	BCD0	Input	Action
0000	0000	0000	10010111	N/A
0000	0000	0001	0010111.	Shift
0000	0000	0010	010111..	Shift
0000	0000	0100	10111…	Shift
0000	0000	1001	0111….	Shift
0000	0000	1100	0111….	Add
0000	0001	1000	111…..	Shift
0000	0001	1011	111…..	Add
0000	0011	01111	11……	Shift
0000	0011	1010	11……	Add
0000	0111	0101	1…….	Shift
0000	1010	1000	1…….	Add
0000	0101	0001	……..	Shift

While the algorithm is extremely simple to use and understand, the way in which it works is slightly more complex. The most important part of the double dabble algorithm to understand is that shifting left is equivalent to multiplying by 2. Since the valid values of a nibble in BCD are 0-9, when the number is greater than or equal to 5, multiplying by two will result in an invalid nibble. The action of adding 3 effectively handles the carrying of the number. For example, if a nibble has a value of 7, shifting, or multiplying by 2 would yield a 14, which is not valid in BCD. It would go from 0111 to 1110. When 3 is added to 7, 10 is obtained. 10 in binary is 1010. When this value is shifted left, the leftmost 1 overflows and the remaining value is 0100, or 4, which forms the units digit of the aforementioned 14. Thus adding 3 can be interpreted as the carry action of the double dabble algorithm.

Before the MCU is ready to perform ADC readings and make calls to the USART peripheral, the peripherals must be setup. For this, the main code makes calls to functions called `TIMER1Setup`, `ADCSetup`, and `init_USART`. These functions simply initialize their respective peripherals, in the same way that `peripheral.begin()` would do in a high-level language. The `Timer1Setup` function performs the following features: firstly, it loads a 0 into the TCCR1A register, which configures the timer in normal mode. Then, it sets the prescaler to 64 by placing predefined T1ps64 into register TCCR1B. This is to trigger the ADC readings approximately twice per second. Finally, the timer is cleared, and enabled by setting the TOIE1 bit in the TIMSK1 register.

The next peripheral that is utilized in this project is the ADC. The ATmega328P includes a SAR ADC, similar to the one built in Project 3.5, although the ATmega328P includes a 12-bit one, instead of a 7-bit one like Project 3.5 does. The ADC converts analog signals and voltages to discrete digital values (see Figure 3).

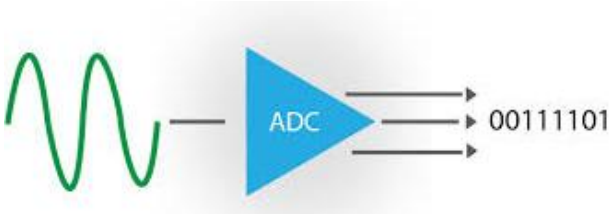


Figure 3. The Matrix Operational Flowchart

The ATmega328P ADC can operate in both in free-running mode, or interrupt-triggered mode. In free-running mode, when the conversion is completed, it automatically runs the next conversion. This means that the ADC is constantly running. While the processor can be doing other things at the same time, it is not ideal from a power consumption standpoint. As depicted in Figure 4, the peripheral never stops running. For the vast majority of applications, the better mode is interrupt-driven mode. This allows the ADC to run upon the overflow of Timer1. In this configuration, when Timer1 overflows, the ADC automatically starts a conversion.

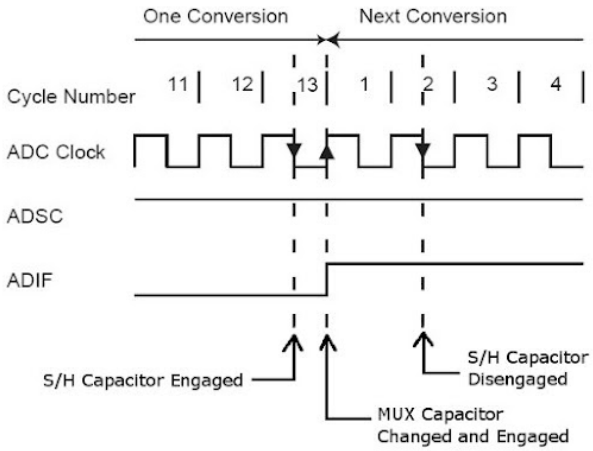


Figure 4. ADC Free-Running Mode

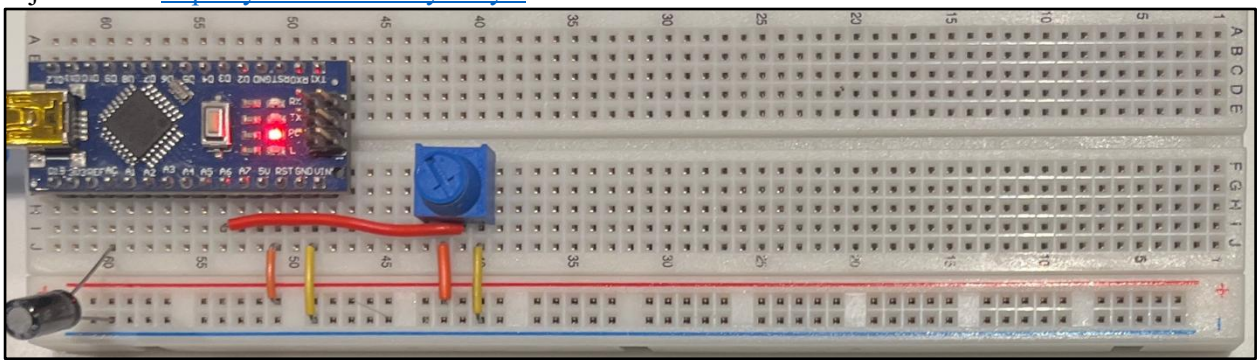
Figure 5. ADC Modes				
ADTS3	**ADTS2**	**ADTS1**	**ADTS0**	**Trigger Source**
0	0	0	0	Free-Running Mode
0	0	0	1	Analog Comparator
0	0	1	0	External Interrupt Request 0
0	0	1	1	Timer/Counter0 Compare Match
0	1	0	0	Timer/Counter0 Overflow
0	1	0	1	Timer/Counter1 Compare Match B
0	1	1	0	Timer/Counter1 Overflow
0	1	1	1	Timer/Counter1 Capture Event
1	0	0	0	Timer/Counter4 Overflow

One major advantage of using assembly language over the standard high-level alternatives is that the processor can perform a second task while the ADC is running a conversion. In a high-level Arduino C, the `analogRead()` function is a blocking function, meaning that the program stops while it is executed. In assembly, the ADC can be started, and the processor can perform another task until the ADC complete interrupt is reached. In this case of the double dabble project, no additional action is performed.

As soon as the ADC completes its conversion, the ADC complete ISR is called. This simply calls the double dabble algorithm, which converts the ADC reading to BCD. Finally, 48 is added to each nibble, or BCD digit, upgrading it to a full byte in ASCII format. 48 is added because it is the offset to get the raw numbers into ASCII-formatted numbers. These bytes are then placed in the UDR0 register, which automatically transmits it over the USART.

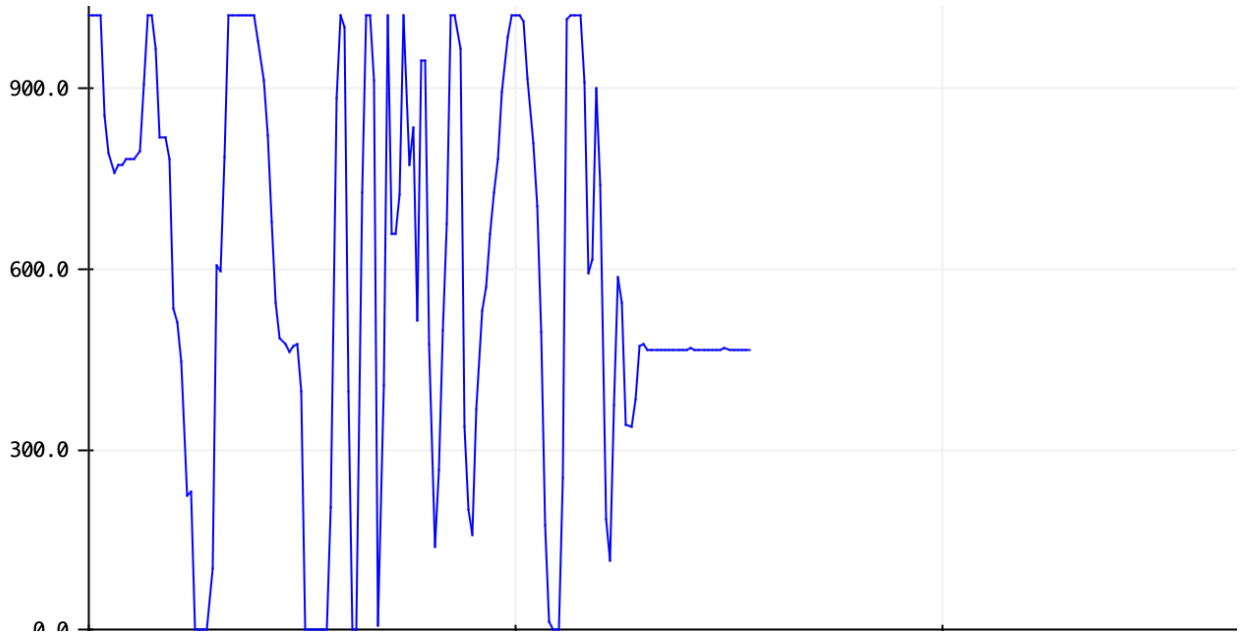
Media

Project video: https://youtu.be/BewBydGeyIs



Double Dabble Project

Serial Monitor as Potentiometer is Spun Back and Forth



Serial Plotter as Potentiometer is Spun Back and Forth

Reflection

This project has taught me several things (perhaps unsurprisingly). Firstly, it taught me how much easier high-level code is than assembly. I knew it was easier, but I have a huge appreciation for high-level code now. Just knowing that my entire 200-line assembly code could have been replaced by 2-3 high-level instructions is both humbling and fascinating. Also, it showed me the somewhat niche use-cases of assembly language. I have heard Mr. D'Arcy talk about the many advantages several times, but this is the first time I am really witnessing and understanding the huge savings. For some reason, I find it cool that you can have the processor doing something while waiting for an ADC approximation to complete. This really does make a lot of sense, but the `analogRead` function always implied that you couldn't, and that the CPU was busy during conversion.

It also gave me a bit of a refresher on time management. I completed the code for the project well before the deadline, so I was ahead of most people in the class in that element. That made me think that I was further ahead than I really was, and writing the report took a lot more time than I had anticipated, leading to a late submission. This was an excellent reminder that projects should be slow burns, rather than several quick spurs. Regardless, this was an excellent project, although I am somewhat sad that assembly is out of the curriculum in both the ACES course, and likely next year in university.