COMP103 (Pre-release)

Microcontroller Programming

Vivatsathorn Thitasirivit
Rev. 0.1 (16 Feb 2023)
https://vtneil.com

Guidebook for IoT and nanosatellites applications using microcontroller.

Notes from Author
This rev. includes basic understanding of electronics knowledge,
hardware communication, abstraction, Arduino programming, pinouts, etc.
Full rev. COMP103 will include more advanced knowledge and techniques. (Soon!)

I. Table of Contents

II. Introduction to Embedded Systems

What is Embedded System?

An embedded system can be thought of a hardware system with software embedded in it for it to function as designed. An embedded system can be either an independent system or a part of larger system. It usually is built on microcontrollers or microprocessors, which is designed to perform specific tasks.

A microcontroller is built for this kind of specific tasks with constraints of low-level design principles and timing.

Characteristics of Embedded System

1. Limited functionality – An embedded system usually performs very specific tasks per designed and repeatedly in some interval of time.
2. Tight constraints – Every design in engineering aspect always has some designing constraints or factors controlling size, weight, power consumption, etc. of hardware and software. An embedded system must be in a form of a board or a chip that can fits in a small space and must be able to process in real-time. Energy consumption is also important in embedded systems.
3. Limited memory – ROM, flash memory, and RAM in embedded system is limited due to its space and energy constraints.

Embedded Systems in Real Life

Around us, there are numerous embedded systems in electrical hardware and Internet of Things (IoT), e.g., temperature control in refrigerators, air conditioners, and heaters; electronic brake systems in modern cars; computer keyboards and mouse; and so many more that you can find them.

Microcontrollers

A microcontroller (MCU/µC) is a small computer on a single integrated circuit (IC) chip. It consists of one or more processors (cores), memory, and programmable input/output (I/O), and usually Analog-to-Digital Converter (ADC). Thus, it is the perfect candidate for applications in embedded systems.

Microcontrollers are programed to receive inputs, write outputs, perform arithmetic calculations, and control. They have relatively lower cost to produce than more complex microprocessors and high-performance CPUs.

Arduino is one of the companies producing development boards for hobbyists and researchers to use with easy-to-use Hardware Abstraction Layer (HAL) interface. It requires only some computer programming knowledge in C and some hardware interfacing knowledge: communication protocols to initially work with it. More features on microcontroller can be used effectively if you have more information on interrupts and low-level components.

III. Hardware Communication

Electrical Layer

When connecting one hardware to another, the electricity is the medium of transferring "data" between them. Electric current directs from higher potential "Voltage" to lower potential. This creates "Potential Difference" or "Voltage Difference" or, in short, just "Voltage."

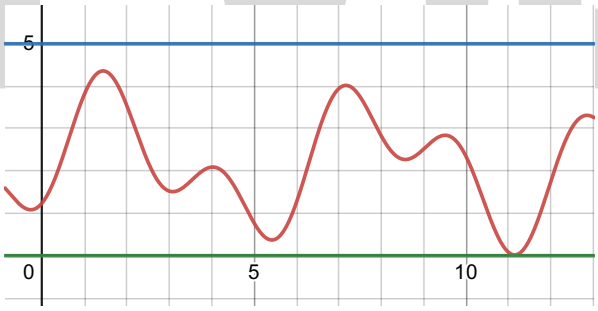
Signals are a form of data sent from one device to another. Usually, the transmitter (sender) sends data through an electrical wire to the receiver end, which the device has some ways of acknowledging it. The data is sent by modulating voltage across that wire of communication. These processes are called "Wired Communication."

There are two distinct ways that a device could modulate the signal for communication: analog and digital.

Types of Electrical Signals

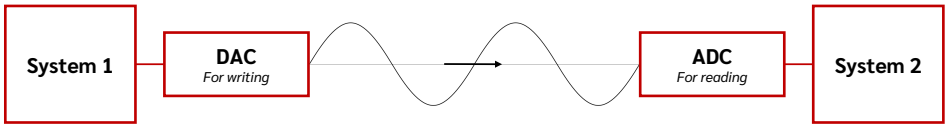
Analog Signal

A traditional way of sending data. Analog signal is a quantity by itself. The amplitude of the signal represents quantity. For example, an analog modulator has a range of voltage output of 0.00 V to 5.00 V. The signal's voltage can be any of values between that range. The signal is, then, converted into a format that computer can understand by "cross-multiplication" (or "rule of three" or "interpolation" or "mapping").



Analog Signal Waveform with range of 0.00 V to 5.00 V

The computer does not implicitly understand what analog signals are as the computers are designed to work in digital format. The hardware responsible for signal conversion is called "Analog-to-Digital Converter" (ADC). There also exists an inverse version of ADC which is "Digital-to-Analog Converter" (DAC). The following diagram illustrates that System 1 convert its digital content into analog signal, sends through wire to System 2 which has ADC to convert the signal back.



ADC Conversion, for example, the computer usually stores data in a string of bits. Each bit can be either high or low (on or off) which translates to "0" or "1." A set of bitstrings are in binary format (base 2) and has 2^n combinations. For example, 1101 has 4 bits; therefore, has $2^4 = 16$ combinations:

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,
1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111.
(Each has value from 0 to 15.)

The number system we use for counting today as a human is in decimal (base 10) with value range of 0 to 9 for each digit.

An analog signal is, by its nature, unsteady, meaning that it can be affected by environmental and electrical noises in a ridiculous percentage. As said, it can be called that "*Analog signal is intolerant to electrical noises.*" As a result, a digital format is created.

An analog input (0.00 V – 5.00 V) is *mapped* to range of 12 bits (0 – 4095). For example, you used a voltmeter to measure the voltage of that line and you get 3.45 V as a result. You can map the result by interpolation:

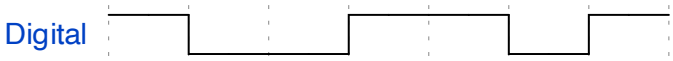
$$\frac{x - x_{min}}{x_{max} - x_{min}} = \frac{V - V_{min}}{V_{max} - V_{min}}$$

$$\therefore x = x_{min} + \left(\frac{V - V_{min}}{V_{max} - V_{min}}\right)(x_{max} - x_{min})$$

The signal reading from computer will be 4,095 * 3.45 / 5.00 = 2,825.

Digital Signal

A more sophisticated way of data representation and communication. The digital signal can be either "0" or "1" based on set voltages. The set voltages are usually at maximum different, e.g., "0" is sent by modulating 0.00 V on the line and 5.00 V for "1." As a result, the digital signal is more robust in term of noise tolerance.



At this level, you are not required to understand how digital signal processing works behind the curtain, but rather understand the concepts and principles of data modulation and representation.

Pulse Width Modulation (PWM)

Coming soon!

Common Wired Communication Protocols

As we understand the concept of how hardware could modulate the signal to transfer data from one to another. Now, we will put "time" into action. If you want to transmit the set of signals, you have many options to do that: you could either send the data in burst and hope that the receiver end can understand or form a standard which every hardware should follow for it to understand each other.

Data can be transferred in either "Parallel" or "Serial" form. Parallel communication enables the transmitter and receiver to communicate many bits simultaneously, but it has drawback of number of wires required and length constraints. Serial, on the other hand, is more versatile as we send the data in "stream." The data is sent on wire contiguously like stream of water in river and only requires at least one wire for data to be transferred. As a result, serial communication is chosen for most applications in external hardware (I/O) communication.

But how would two devices know how fast it should send and how fast it should receive? There are two classifications of communication based on "clock": "Asynchronous" and "Synchronous" communication.

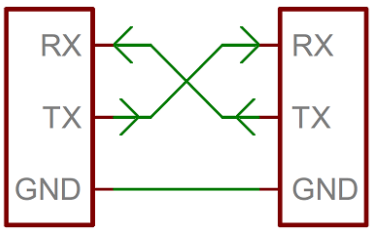
Universal Synchronous/Asynchronous Receiver/Transmitter (USART)

Universal Asynchronous Receiver/Transmitter (UART) is a subcategory of USART which does not require clock to operate. UART is just a protocol used for standardizing serial communication. There are also higher-level standards implementing UART in them, e.g., RS-232, RS-485, etc.

UART requires only 2 – 3 pins or 4 – 5 pins to operate between 2 devices:

1.	Transmitter Pin (Tx)	Data output pin
2.	Receiver Pin (Rx)	Data input pin
3.	RTS#	Ready to send (Transmitter requests to send)
4.	CTS#	Clear to send (Receiver responds that it is clear)
5.	Common GND	For voltage references between different system

Logically saying, the transmitter pin Tx of one device should be connected to the receiver pin Rx of another device and vice versa as follows.

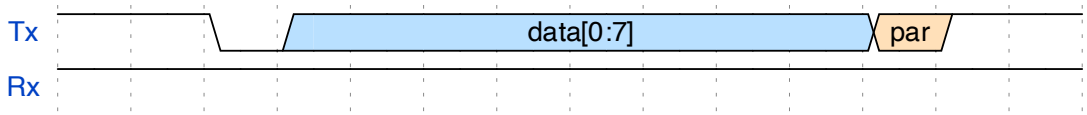


UART Wiring (Source: SparkFun)

The UART protocol has standard for data payload. In default situation where there is no data transmission happening, the communication line is always HIGH. When

the transmitter wants to send data, it pulls the line LOW first, then send *n*-bit data with a parity bit for error detection. Lastly, the transmitter will reset the communication line by pulling the line back to HIGH. Number of bits depends on the standard.

The following figure shows timing diagram (waveform) of UART transmission. Sender transmit data on Tx line while the Rx line is clear.



How does UART know which speed to transmit/receive? The programmer must predefine the "baud rate" which is a preset speed, e.g., 9600 baud, 115200 baud, etc. UART sends data in a small payload to prevent clock shifting. On two different devices, the clock speed might not be equal (at least for over long period of time), so, to prevent clock skewing, the data is sent in burst. This asynchronous scheme already has some drawbacks in terms of data speed and clock shifting, so synchronous scheme which uses clock to help keep track of how many bits the transmitter has transmitted, which are more complex standards.

UART has peer-to-peer topology, so if you have many devices that need to communicate with each other, the number of wiring quadratically increases. Also, UART sends data[0:7] in low order bit first: D0, D1, D2, ..., D7.

Serial Peripheral Interface (SPI)

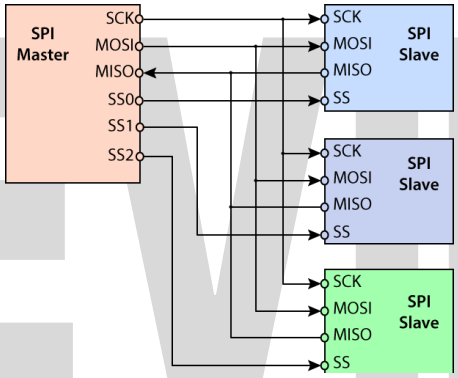
SPI is a communication protocol upgraded from ordinary USART. It includes clock and adds master-slave to it. SPI is synchronous and full duplex, just like USART.

Master-Slave configuration allows one device to be master (a hub taking control of the communication line and managing which slave device should be communicating with the master device.

SPI uses at least 4 – 5 lines for communication:

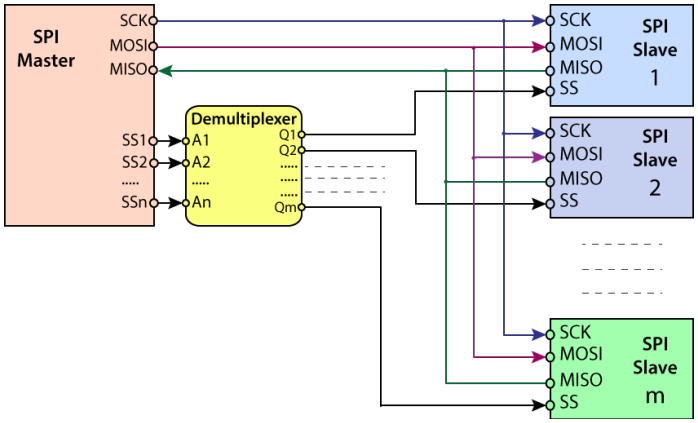
1.	SCK	Clock Pin
2.	MOSI (SDO)	"Master out, Slave in": Master writes to slave.
3.	MISO (SDI)	"Master in, Slave out": Slave writes to master.
4.	nSS or nCS	Active low slave select for selecting devices. If there are multiple slave devices, there will be the same number of nSS pin as number of slave devices.

The following figure shows the configuration of 1 master and 3 slaves.



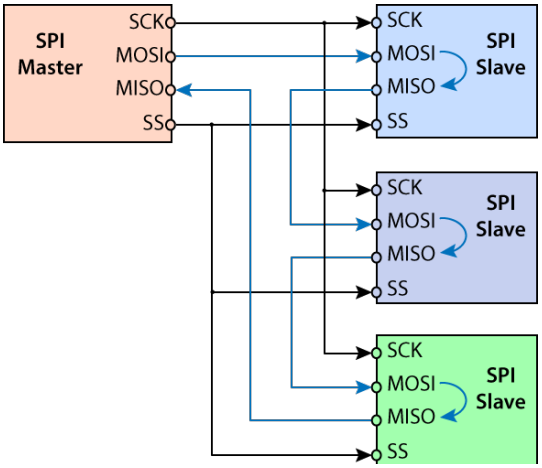
SPI with 1 Master 3 Slaves configuration (Source: DLNWare)

For a n to 2^n slaves mapping, you can apply a demultiplexer (decoder) for nSS pins as follows.



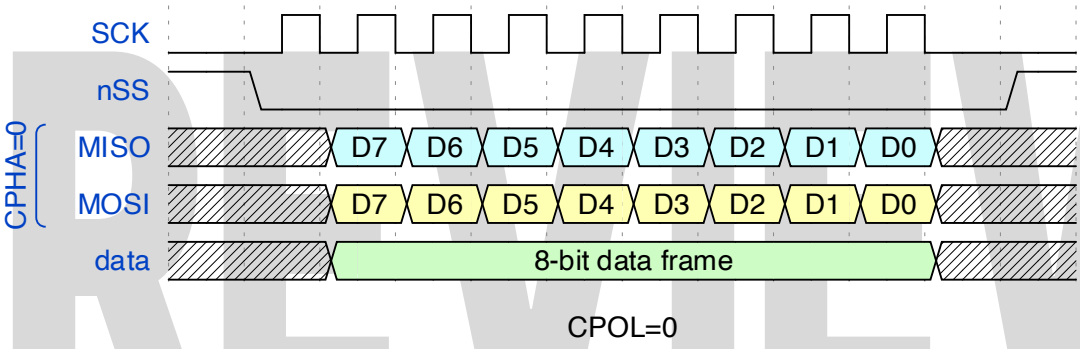
SPI with decoder configuration (Source: DLNWare)

Supported SPI devices can also be daisy-chained as follows.



SPI daisy chain configuration (Source: DLNWare)

Timing Diagram of SPI:



In SPI, the data will latch (read) on rising edge of the clock and will change (unstable) on falling edge. MISO and MOSI lines are data path. SCK and nSS lines are control path which allow the slave to know when to send or receive.

SPI has constraint of number of devices per pin connection like UART, so you cannot connect many devices with SPI. Some protocols are designed to support SPI implicitly, one of which is SD protocol which you can directly connect SD card's pins to the SPI pins on the board.

Inter-Integrated Circuit (I^2C)

I^2C is synchronous, half-duplex, and multi-master-slave communication protocol based on “Bus.”

A Bus is a form of utilizing one wire as a medium for every device to communicate. It can be visualized as there is one bus on the long road where one same group of people can get on at a time. Until the bus is free, it can be occupied by another group of people. In computer, Bus has the same idea of data communication. Suppose there are three devices on the bus: Dev0, Dev1, and Dev2. While Dev0 is using the bus to communicate with Dev2, Dev1 cannot use the bus because the bus is currently occupied. Once Dev0 releases the Bus to be free, Dev1, then, can use the bus.

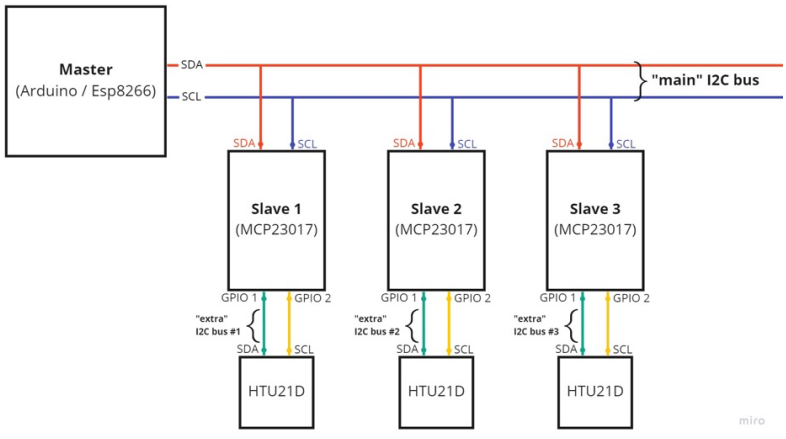
I^2C has many characteristics:

1. Software-addressable – each slave device has its own preset address.
2. Collision detection
3. Serial, 8-bit, Bidirectional, 4-mode
4. On-chip filtering circuit

I^2C only uses 2 – 3 line to communicate with every device on the same line (bus):

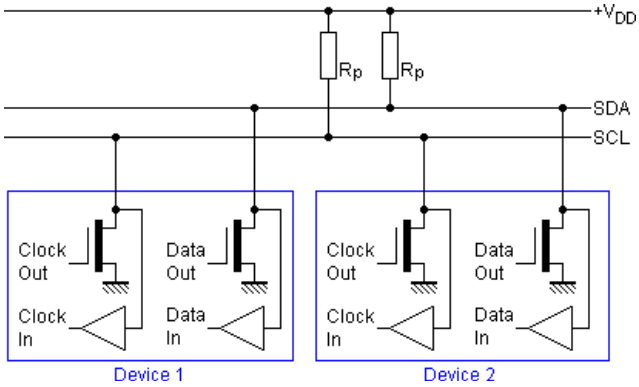
1.	Data Line (SDA)	Bus where data lives
2.	Clock Line (SCL)	Bus where clock operates
3.	Common GND	For voltage references between different system

You can connect every device in parallel as the line standard is Wire-AND connection. Figure below illustrates parallel connectivity of I^2C (Note that the figure is missing a pull-up resistor, usually 4.7 kΩ).



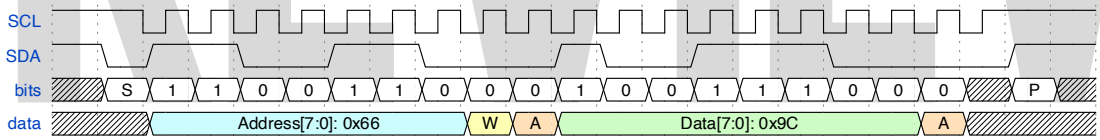
I^2C parallel connection with 1 master and 3 slaves (Source: Unknown)

The line is connected by a pull-up resistor connecting to Voltage source (V_{DD}) because the transmitting mechanism of the device uses “open-drain” using 1 MOSFET rather than “push-pull” using 2 MOSFETs. Open-drain means the circuit is “Open” by default and “Drained” when powered. So, if the transmitter wants to transmit data, it only has to pull down (drain) the line to make whole line LOW. After it is done transmitting, it can release the line by opening MOSFET, so that the voltage source does not drain and, subsequently, make the line HIGH.



I²C open drain on the bus SDA and SCL (Source: EENG 383 Course)

Data transfer in I²C is more sophisticated than in UART and its engineering aspect is quite fascinating. By default, SDA and SCL line are HIGH. If the master wants to read/write data from/to the slave, the master will pull SDA down to LOW and begin transmitting a data payload (frame) to the SDA line while starting modulating clock on SCL line. The following timing diagram shows I²C data frame (sends high order bit first).



The data payload consists of the following.

1. Start bit (by pulling down SDA)
2. 7-bit Address (MSB first)
3. R/W Code (HIGH for Read, LOW for Write)
4. NACK (Not acknowledge) from slave: LOW means ACK, HIGH means NACK
5. 8-bit Data frame (MSB first)
6. NACK (from slave if write, from master if read)
7. 1 clock wait
8. Stop bit (by releasing SDA) for resetting

If NACK is raised, it could mean any of these conditions:

1. If NACK is raised after Address, R/W, then no slave responded.
2. If NACK is raised after Data in W mode, then slave did not accept data.
3. If NACK is raised after Data in R mode, then master did not want data anymore.

IV. Wireless Communication

Coming soon!

Wireless Communication Protocols

V.Common Peripheral Devices

Coming soon!

Status Indicator

Light Emitting Diodes (LEDs)

Buzzers

Display and LCD Display

Barometric Sensors

Analog Sensor

Breakout IC

Geolocation Sensors

Light Sensors

Force Sensors

VI. Data Transmission Design

Coming soon!

VII. Program Design

Before getting to hardware selection, connection, and programming, you must go through the processes of missions and requirements designs. The program is built based on available factors and constraints.

Suppose you have gone through all those processes: missions, hardware, and constraints. You, as a programmer, should design your programs before start coding in any specific languages required for the project. These steps will help you and other programmers working in/after your project maintain the integrity and correctness of the program.

Steps and Methodology

1. Understand the program – you need to clearly and fully understand how your program should operate according to the designed missions and constraints. If you do not completely understand what you are about to do, your source codes and program will end up in a *KABOOM*.
2. Create a Model – you can write a state diagram, a flowchart, or steps of operations with the method you are comfortable to visualize how the program should operate under predetermined conditions.
3. Develop Test Data – the test data is optional for some systems and required for some systems, preferably systems with data transmission which you would have to disassemble the program into parts for debugging.

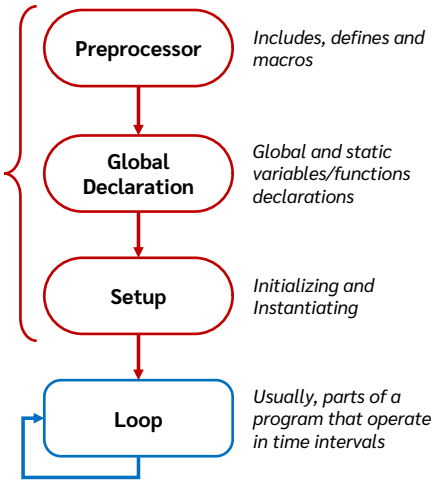
Program Structure

A program in a system is something that takes **input** data, **process** them with some logic, and **output** the final product. Sometimes, you would find a feedback path for system response. The most difficult part is designing the process of the program to satisfy and optimize output based on inputs.

Program Flow: Setup-Loop Variant

In microcontroller programming, the scheme of operation is usually: 1. setup phase and 2. loop phase.

The scheme is standard across various designers and manufacturers. Because of that, the implementation is very similar across different brands. Arduino's Hardware Abstraction Layer (HAL) implements the scheme and create a user-friendly programming interface.



1. Preprocessor – this is where you can include other source code files and define some macros for you to use in the program. This section is usually not the core of programming, but an essential part in compiler-based programming, i.e., C.
2. Global Declaration – all global and static variables and functions will be compiled and "put" them in the memory first. Global variables can be accessed by all parts of the program in the source file. Usually, this approach is not a good practice, but in limited memory and time-sensitive applications in embedded system, and for ease of usages, global declarations are quite common to do. Nevertheless, the programmer, you, must be careful when programming in terms of optimization, naming, and scoping.
3. Setup – in here, objects, e.g., sensor HAL classes, are "instantiated" and "initialized" for later usages in the program. For example, A BME280 sensor object is initialized on I^2C line with address of 0x76 on default Arduino I^2C line. You can insert any program you want the microcontroller to operate **once** and only once after the booting process. The program will not return to the setup phase unless you reset, reboot, or turn on.
4. Loop – this is where the exciting parts begin. A loop phase is a section of a program where it runs forever. Once a section of the program in the loop ends, it jumps to the beginning of the loop, and it repeats forever.

Examples

Potentiometer & PWM LED

A program that reads analog input from potentiometer determining LED brightness. This one has clear I/O and somewhat simple calculation in the process.

1. Input: Analog value (mapped) ranging from 0 to 4095 (12-bit)
2. Output: Analog output (PWM) capped at 255 (8-bit)

To design a process, you must know what the program should do. In this case, the program must map larger 12-bit input to write the pin as analog output with smaller bits (8 bit). Now you know what to do as you learned value interpolation mapping.

Blink Speed Control

A program that blinks an LED every <interval> seconds. If a user presses a button, the interval changes according to the cycle, e.g., the cycle is 0.2 s to 0.5 s to 1.0 s and back to 0.2 s and so on.

1. Functional aspect: The LED blinks every <interval> seconds.
2. Input: User's button pressing
3. Output: LED blinks with new interval of <next_interval> seconds.

To design this program, the concept of Finite State Machine (FSM) can be directly implemented into the program. In this case, there are 3 program states: S0, S1, and S2. Each state has its own interval value. When the user presses the button, the state should change (cycle). I.e., S0 -> S1; S1 -> S2; S2 -> S0. You can implement this program logic in the Arduino with C using many methods: state switching, array cycling, etc.

Temperature Warning

A program that reads data from a temperature sensor. If the temperature is over 30 degrees Celsius, the LED will blink with 0.5 s interval. If not, the LED will stay off until the temperature threshold is reached.

1. Input: Temperature value data from sensor
2. Output: LED Blinks or not?

To design this program, recall that there is a setup-loop scheme. The idea is that the program will check whether the temperature exceeds set value or not repeatedly using a conditional statement in an infinite loop. We then write the program so that if the condition passed, start blinking an LED.

Solution Approaches

Programming a computer is a way of problem solving. Programming a microcontroller also utilizes such field of knowledge to “make things work.” There are many approaches to a single problem you can implement; each has its own advantages and drawbacks which you, the programmer, will be the person to decide. A solution to one problem might not work for other problems, but a part of the ideas could potentially help you solve them.

There is no ultimate all-in-one solution for every problem. The programmer must analyze and go through the process to create an algorithm or a solution to the problem.

There are many kinds of computational problems in this world: sorting, searching, graphs, isomorphisms, and many more. The solution in microcontroller programming context is that which features of the programming language can be used or applied to ease the logic.

For example, a loop in the language can help you reduce source code repetition, and help you iterate through arrays. A conditional statement and jumping help increasing performance of the program by optimizing the logical parts.

VIII. Basic Arduino Programming

Arduino IDE is an integrated development environment for programming microcontroller supporting Arduino interface, e.g., Arduino Pro Mini, Arduino UNO, Teensy, etc.

Arduino Hardware

An Arduino board is a development board consists of many passive and active components, one of which is the microcontroller. In this example, I will introduce you the ATMel AVR ATmega328P, which is the most basic microcontroller you can find in the market with minimal functionalities. It comes in Arduino UNO R3, Arduino Pro Mini, and Arduino Nano, etc. development boards. These three boards differ in number of pinouts, size, and ports. You can build systems with some fundamental functions at low speed and at low performance with them. In this guide, Arduino UNO R3 will be used because of its versatility for learning.

Arduino UNO R3 Pinouts

Arduino UNO R3 Pinouts (Source: Arduino Official, Feb. 2023)

Basic C Arduino Programming

The microcontroller can be programmed by many languages: Assembly, C, C++, MicroPython, etc., but C is widely used in this field specifically because it has high-level syntax and logic but with low-level direct access to hardware. Consequently, the codebases of embedded systems in C are widespread and large. You can find nearly every type of project you can think of on the internet from a blinking LED to an actual source code of small-scale rockets.

Source file structure

```
void setup() {
  // put your setup code here, to run once:
}
void loop() {
  // put your main code here, to run repeatedly:
}
```

Program delay

```
delay(1000); // delay for 1 second (put this in a function)
```

Setting pin mode

```
pinMode(13, OUTPUT); // set digital pin 13 to output
```

UART Communication (via USB)

```
void setup() {
  Serial.begin(115200); // initialize UART with baudrate 115200

  Serial.println("Hello world!"); // send message through UART once
}

void loop() {

}
```

I²C Communication (Read from BME280)

Coming soon!

Arduino Programming Examples

More of examples can be accessed from Arduino IDE > File > Examples.

LED Blink

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);  // Set pin voltage to HIGH (on)
  delay(1000);             // Wait for 1 second
  digitalWrite(13, LOW);   // Set pin voltage to LOW (off)
  delay(1000);             // Wait for 1 second
}
```

Read Analog Pin's voltage (mapped)

```
void setup() {
  Serial.begin(115200);
}

void loop() {
  int sensorValue = analogRead(A0); // Read analog from pin A0

  float voltage = sensorValue * (5.0 / 1023.0); // Maps value range

  Serial.println(voltage); // Show through USB UART
}
```

IX. Advanced Programming Schemes

You can superpower your system with more advanced techniques in programming the microcontroller to increase processor efficiency and performance. These techniques require more advanced knowledge on hardware level, which, in this section, will be broadly discussed.

This section is coming soon!

Non-blocking Delay (Pseudo-asynchronous)

Polling

Interrupts

Direct Memory Access (DMA) Interrupts

Direct Port Manipulation

X. References (As of now)

Chris M. Finen. Communications and Protocol Basics.