**THE EMBEDDED HARDWARE SYSTEMS COMPENDIUM**

By Bruno Hinojosa

Logo

Description automatically generated

**Preface**

*This compendium is a compilation of all concepts and systems that the STARX Embedded Hardware sub-team has used, along with explanations to help members learn or revisit when needed. This compendium will change as new systems are introduced and old ones are replaced. If you find any corrections or errors that need to be fixed, or if you wish to add something new to this document, please do not hesitate to ask the current team lead to make such adjustments.*

*As one of the previous Embedded Hardware Systems once said to me “each member and each lead brings their own spin to the current suit”. And with that sentiment don’t be afraid to change the system we have now, as long as you are moving forward and commit to making the suit to yield good results there should not be any creative restrictions from any STARX member and remember this club is nothing but the effort you put in.*

**Table Of Contents**

Introduction ------------------------------------------------------------------------ 5

Current System ------------------------------------------------------------------ 6

Fundamentals -------------------------------------------------------------------- 9

Arduino Fundamentals --------------------------------------------------------- 9

Arduino Pins ------------------------------------------------------- 10

Arduino Program Structure -------------------------------------- 12

UltraMotion Linear Actuator Fundamentals -------------------------- 13

Configuration ------------------------------------------------------- 13

Electrical interface and Pinouts ---------------------------------- 15

Interrupts Fundamentals ------------------------------------------------- 18

Hardware interrupt ------------------------------------------------- 18

Software interrupt -------------------------------------------------- 18

Pulse Width Modulation Fundamentals ------------------------------- 19

Sending a PWM with Arduino ----------------------------------- 21

Receiving a PWM with Arduino --------------------------------- 21

PID Fundamentals -------------------------------------------------------- 25

Closed loop systems ----------------------------------------------- 25

Proportional response --------------------------------------------- 26

Integral response --------------------------------------------------- 26

Derivative response ------------------------------------------------ 27

Accelerometer/Gyroscope Fundamentals ----------------------------- 29

Communication Protocol Fundamentals ------------------------------- 29

Serial Communication Protocols --------------------------------- 30

Transmission modes in Serial Communication ---------------- 30

Synchronous Serial Interface ------------------------------------- 31

Asynchronous Serial Interface ----------------------------------- 33

References ------------------------------------------------------------------------ 35

**Introduction**

STARX is dedicated to designing and building a strength-augmenting robotic-exoskeleton suit. Exoskeletons are an emerging technology and are starting to become used for several unique applications, like factory work and emergency personnel. Although the main goal of the STARX team is to design and build an exoskeleton suit, the suit is targeted toward the ACE Competition. ACE is a cross-university competition aimed at testing the functional capabilities of exoskeletons designed and built by their respective universities where teams can test the actual real-world potential of their designs. The competition is designed around elements of the Candidate Physical Ability Test (CPAT), which assesses the physical abilities of entry-level firefighters. This provides a reliable test of real-world physical tasks which a powered exoskeleton would likely encounter.

We are on the third iteration of STARX’s exoskeleton suit. I joined STARX when the suit we have today was just a concept we wanted to try. The old version of the suit consisted of pneumatic muscles instead of linear actuators, and the control system was simple: it used potentiometers to measure the angle of flection of the pilot’s leg which determined when to actuate the pneumatic muscles. After changing to linear actuators, the team decided to also change the potentiometers for EMG sensors. These sensors measure small electrical signals generated by a muscle once you move them. This change came with its own challenges when it came to reliability and communication, so this system was also scratched. There was also an attempt to introduce machine learning to the system alongside the EMG sensors but since the only one who really knew how to program and use the machine learning algorithms was the previous Embedded Hardware Systems lead, who is no longer on the team. The machine learning code was not used in the third iteration of the suit.

The most up-to-date version of the suit’s control system uses gyroscopes, linear actuator feedback, and serial communication to be able to calculate the desired linear speed of the actuators. The code makes use of PI controllers to converge the current speed of the linear actuator to its desired speed. The suit will also compare the angle of orientation of the suit’s legs and the pilot’s legs to mitigate a propagating error from the first PI control. This feature is still being developed, however.

**Current System**

The current system is not that complicated to follow once you understand each piece or subsystem employed in the making of the suit. We have three different devices that oversee delivering, interpreting, and giving information to make the suit move: The Arduino, the linear actuator, and the gyroscope. It’s easier to look at the suit one leg at a time and each leg can be divided into two sub-systems: the hip and the knee control systems.

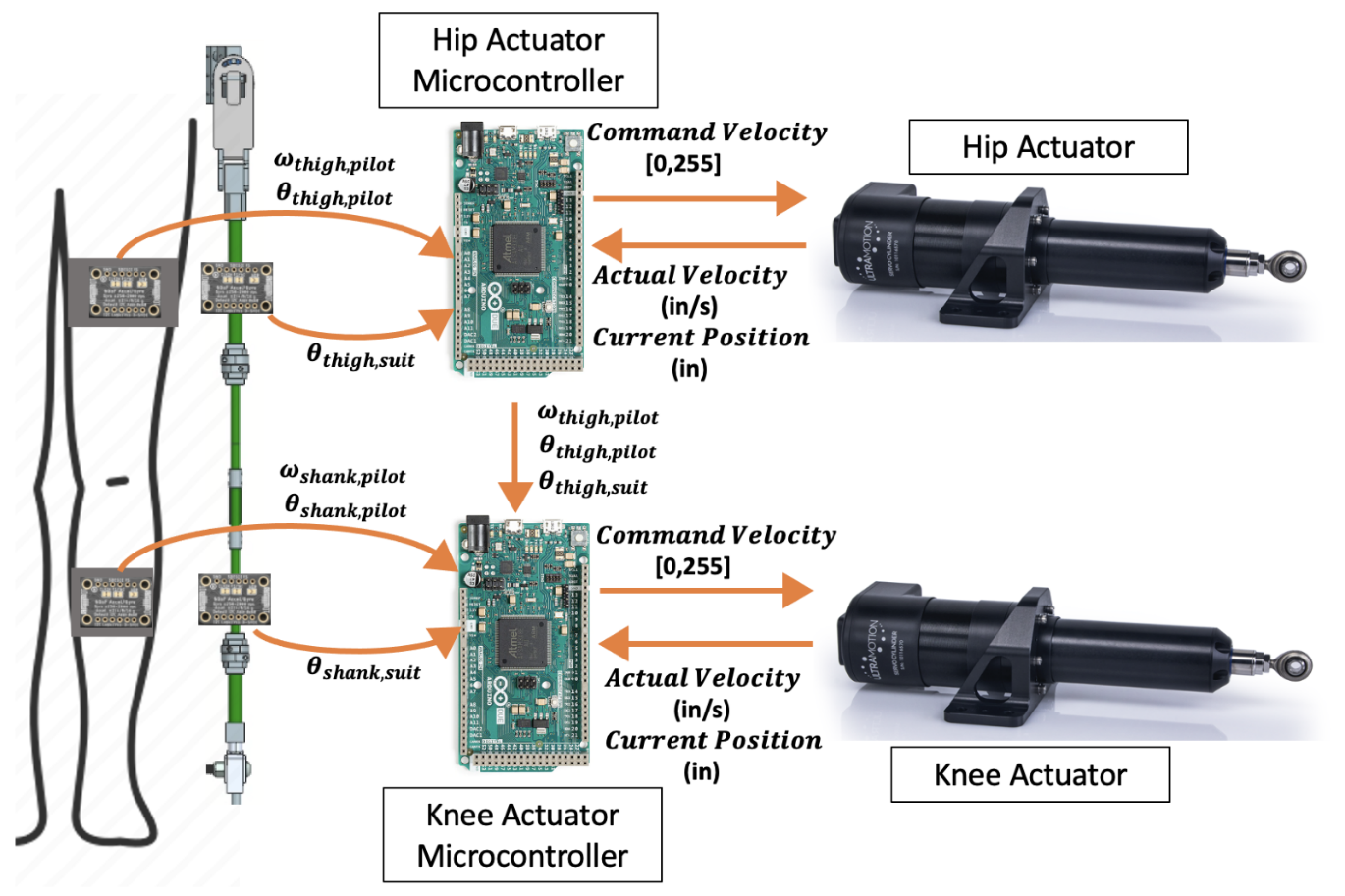


Figure 1: One leg system I/O overview

The Arduino is the main processing unit, and it interprets all the readings of the sensors on the suit. The gyroscopes return angular velocity and acceleration values from the suit and the pilot, and the linear actuators return the actuator length/stroke and actuator speed. We use the angular velocity retrieved from the gyro on the leg of the pilot to determine the desired linear actuator velocity. To ensure that the real actuator velocity matches the calculated desired actuator velocity, we use a PID control algorithm called *LoadCompensator()* in the code. What differentiates the knee sub-system, and the hip sub-system is the fact that the knee requires the hip angular velocity information to compute the knee’s linear actuator velocity. Because of this, the hip Arduino needs to send the gyro readings to the knee Arduino. We use an asynchronous communication protocol called UART to accomplish transmit this data. On top of this, the Arduino can trigger the suit to go to a “standing position” with the push of a button. When the button is pressed the Arduino turns off the load compensator PID and turns on the *StoppingPID()*. The *StoppingPID()* is another control algorithm that ensures that the linear actuator length matches the desired length that will set the suit to a standing position. Once, the suit reaches the standing position the linear actuator locks into place.

Although our design accomplishes the different goals we set out for ourselves, it is not a perfect system. The code has some variables, constants, and functions that have become redundant due to the many changes and rewrites. The function *PIDtoggle()* is a good example of this. Although the function runs with no errors, it is never called in the main code giving it no reason for it to be there. There also might be better and more clever ways to perform the same actions but with different code.

Diagram

Description automatically generated

Figure 2: Hip sub-system code flowchart

**Fundamentals**

This section introduces several concepts including digital input/output pins, pulse width modulation, interrupts, and serial communication, among others.

**Arduino Fundamentals**

We chose Arduinos to perform the control of the many systems of the suit because of how easy it is to program and troubleshoot with the Arduino boards. Arduinos also offer a good range of applications due to the variety of pins that the boards have to offer.

Arduino boards can read inputs – light on a sensor, the inclination of a gyroscope, or the feedback of a system – and turn it into an output – activating a motor and turning on an LED. You can tell your board what to do by sending a set of instructions to the microcontroller on the board using the Arduino IDE software.

The Arduino boards we used on the suit were chosen because they are relatively inexpensive and easy to use. Also, there are many resources and libraries online when it comes to programming and wiring the Arduino boards. Each Arduino board has its own features, and they are different from each other not only on the hardware level but also on the software level. That’s why anytime you program a new type of Arduino board for the first time you have to download/update the board libraries on the *Board Manager* window. This ensures that the ‘built-in’ libraries and functions like *digitalWrite()* can actually perform like they are supposed to on the specific board you have connected.

Text

Description automatically generated

Figure 3: example code using the built-in library functions

It is also worth noting that the Arduino Due uses the *Atmel SAM3X8E* microcontroller. This microcontroller chip is completely different from the more common boards (Uno, Mega, Pro Mini, Leonardo), making some of the ‘built-in’ libraries and functions inoperable.

There’s no particular reason why we use the Arduino due particularly. The board itself has many extra pins that other boards don’t have. The due also has bigger memory. This was the main reason why we chose this board back when we still used machine learning for our control system. The extra memory allowed us to upload the learning algorithm to the board. It’s worth noting that the Arduino Mega would also be able to control our current suit.

**Arduino Pins**

All Arduinos have a variety of programmable *Pins* responsible to read inputs from the outside world or sending signals to the outside world. A picture of the Arduino Due pinout can be seen in Figure 4.

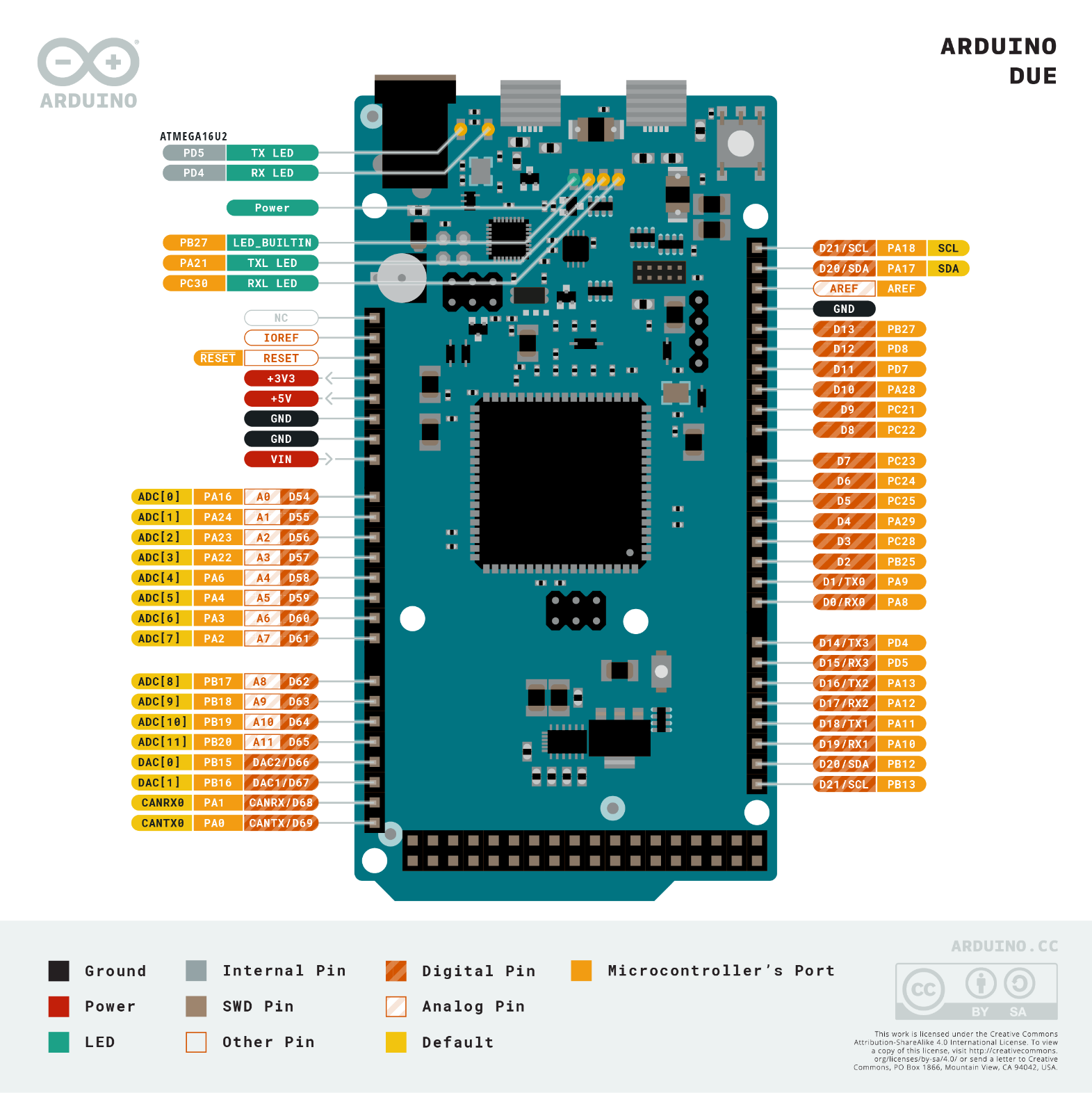


Figure 4: Arduino Due Pinout

Each Digital I/O pin on the Due can be used as an input or output, using *pinMode(), digitalWrite(),* and *digitalRead()* functions. They operate at 3.3V and each pin can provide a current of 3.3A or 15mA, depending on the pin, or receive a current of 6mA or 9mA, depending on the pin.

In addition, some pins have specialized functions:

**Serial: 0 (RX) and 1 (TX)**

**Serial 1: 19 (RX) and 18 (TX)**

**Serial 2: 17 (RX) and 16 (TX)**

**Serial 3: 15 (RX) and 14 (TX)**

These pins are used to receive (RX) and transmit (TX) TTL serial data (with a 3.3V level). Pins 0 and 1 are connected to the corresponding pins of the ATmega16U2 USB-TTL Serial chip.

**PWM: Pins 2 to 13**

Provide 8-bit (0 to 255 range in hex) PWM output with the *analogWrite()* function.

**TWI 1: 20 (SDA) and 21 (SCL)**

**TWI 2: SDA1 and SCL1**

Support TWI (Two-Wire Interface) communication using the Wire library. SDA1 and SCL1 can be controlled using the wire1 class provided by the Wire library only after adding internal pull-up resistors. While SDA and SCL already have internal pull-up resistors, SDA1 and SCL1 do not.

**Analog Inputs: pins from A0 to A11**

The Arduino Due has 12 analog inputs, each of which can provide 12 bits of resolution (i.e., 4096 different values). By default, the resolution of the reading is set at 10 bits (i.e., 1024 different values), for compatibility with other Arduino boards. It is possible to change the resolution of the ADC with the *analogReadResolution()* function. The Due’s analog input pins measure from ground to a maximum value of 3.3V. Applying more than 3.3V on the Due’s pins will damage the SAM3X chip.

**DAC1 and DAC2**

These pins are Digital-to-Analog converters (ADC) that provide true analog outputs with 12-bits of resolution (4096 levels) with the *analogWrite()* function.

**Arduino Program Structure**

The Arduino software structure consists of two main functions – *setup()* and *loop().* The setup function is the first function that is called when the Arduino board is powered or reset, and it will only run once.

Graphical user interface, text, application

Description automatically generated

Figure 5: Setup() and loop() functions of an Arduino sketch

The setup function is used to initialize variables, pin modes, call setup functions from libraries, etc. Note this function cannot receive parameters nor return any values. Note: variables and objects can also be created above/before the void setup or the void loop.

Once all the code under the setup function has run, the microprocessor then calls the loop function. The loop function, as the name suggests, is an infinite loop that runs indefinitely. This function is used primarily to read signals from the I/O pins and/or output signals from said I/O pins. Note that the programming in Arduino does not stop with these two functions. Although they dictate the program’s structure, it’s possible to expand the functionality of the code by creating local functions and/or adding libraries to your Arduino sketch (code).

**UltraMotion Linear Actuator Fundamentals**

The Ultra Motion linear actuator is a high-performance linear servo system comprised of a rod-style actuator, a configurable brushless DC (BLDC) motor controller, and the Phase IndexTM. To use the actuator a user must configure the settings for defining many attributes including performance, characteristics, software limits, and output information, among other configurable parameters. After configuring the linear actuator, the only things it needs to operate are the input power (battery or power source) and the command input set in the configuration process. Common command inputs include CANopen, serial, analog, and stepper signals. I strongly recommend that you read or at least get familiar with the UltraMotion servo cylinder manual to get a deeper understanding of how this machine works. Specifically, pages 8-9 (Actuator control systems), 25 (Actuator pin colors), 34-38 (Actuator pins), 39-43 (Actuator and config setup), 44-58 (All the config settings, don’t really need to know), 66-70 (PID graphs), 91 (Conversion Tables)



Figure 6: Ultramotion A1 series linear actuator

**Configuration**

To configure the linear actuator the machine needs to be powered and the USB cable needs to be connected to your computer/laptop (apple products are not supported by the linear actuator firmware, so please use a Windows device to configure the linear actuator). Once the actuator is connected to your device open the CONFIG.TXT file. By changing the values of the different parameters on this file you can set any configuration the actuator has to fit your needs. After performing the changes save the file, disconnect the USB cord from the actuator, and power it off.

Graphical user interface, text, application

Description automatically generated

Figure 7: Snip of the CONFIG file

Currently, the linear actuator is configured in *Toggle* (Simple On/Off for Extend/Retract) control. In *Toggle* mode, the Servo cylinder operates with the simplicity of a brushed DC control system: the actuator simply extends for as long as it is given an “extend” signal to its digital input and retracts likewise. This mode also allows for configurable acceleration and velocity for smooth, controlled motion, and the output speed can be controlled by using analog inputs.

When plugging the linear actuator into your computer there will be another file called HARDWARE.TXT. the HARDWARE.TXT contains hardware-specific information not intended to be edited by the user unless suggested by one of the Ultra Motion application engineers after consultation. This file is specific to each actuator and cannot be copied between multiple linear actuators (See figure 8 below).

Graphical user interface, application, Word

Description automatically generated

Figure 8: HARDWARE.TXT example

**Electrical Interface and Pinouts**

Diagram

Description automatically generated

Figure 9: Pinout for A1 series Actuator

Table 1: Pin numbers, functionality, and standard cable wire colors for A1 series actuators

|  |  |  |
| --- | --- | --- |
| No. | Pin for “N” Controller | Flying Lead Color  (CBL-A1DEV) |
| 1 | ANI | WHITE |
| 2 | GND | BROWN |
| 3 | TX | GREEN |
| 4 | DIO1 | VIOLET |
| 5 | IN2+ | GRAY |
| 6 | IN1+ | BLUE |
| 7 | V+ | RED (Spade Terminal) |
| 8 | V+ | Red (Spade Terminal) |
| 9 | ANV | RED |
| 10 | GND | BLACK |
| 11 | RX | YELLOW |
| 12 | DIO2 | ORANGE |
| 13 | IN2- | PINK |
| 14 | IN1- | TAN |
| 15 | GND | BLACK (Spade Terminal) |
| 16 | GND | BLACK (Spade Terminal) |

The Compendium will only cover the Pins we are currently using in the suit. If you’d like more information on the other pins and what they do, please read the UltraMotion servo cylinder manual.

**Analog Voltage Input (ANV)**

This pin has a -10 VDDC to +10 VDC voltage rating and it can be grounded to any GND. It is used to receive an analog voltage signal, which can be used to control actuator position or control either speed or torque depending on the control mode. It’s worth mentioning that the ANV pin can read raw PWM signals from the Arduino.

**Optically Isolated Digital Inputs (IN1+, IN1-, IN2+, IN2-)**

The linear actuator’s digital inputs are optically isolated, meaning that they are electrically isolated from the rest of the circuit and therefore do not share a common ground. This is to provide optimal noise rejection. It is acceptable to connect In1- or IN2- to actuator ground to simplify wiring, but this will remove the benefit of the noise rejection from optical isolation.

Diagram

Description automatically generated

Figure 10: Schematic for Optically Isolated Digital Inputs.

In Figure 10, the resistor “R” is a current limiting resistor necessary when using voltages that exceed 7V on the Classic Controller linear actuator line.

Table 2: Input states and behavior

|  |  |  |  |
| --- | --- | --- | --- |
| Behavior | Input 1 (IN1) | Input 2 (IN2) | Notes |
| Stop Mode Active | 0 (off) | 0 (off) | - |
| Extend Active | 1 (on) | 0 (off) | Extends while active and will stop if spMax is reached. |
| Retract Active | 0 (off) | 1 (on) | Retracts while active and will stop if spMin is reached. |
| Dynamic Brake | 1 (on) | 1 (on) | - |

**General Purpose Input/Output (DIO1 and DIO2)**

The linear actuator has two GPIO pins which can be used for a wide variety of applications. It is important to note that DIO1 and DIO2 are not optically isolated, meaning that they may be more prone to inject inadvertent noise into the system. The GPIO pins can be configured to be used as a high impedance digital input, or to output any of the following:

* +5 VDC supply (up to 250mA)
* Ground (sink up to 250mA)
* 1kHz PWM waveform proportional to a variable set by ioPWM1 and ioPWM2 – PWM Output Signal Type for GPIO pin 1 and 2 respectively (ioPWM1 and ioPWM2 are variables that can be set in the CONFIG.TXT) file. The output variable includes position, velocity, force, or bus voltage.
* Limit switch signals (trigger at specified travel distances). To use this feature, use the masked status register see page 65 on the linear actuator manual
* Output the result of the masked status register. See page 65.

The GPIO pin configuration we use on the suit is the 1kHz PWM Output Signal type, and their respective output variables are velocity and position.

**Interrupt Fundamentals**

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an *Interrupt Service Routine* (ISR). ISRs tell the microprocessor or microcontroller what to do when the interrupt occurs. We use interrupts in the suit to read PWM signals coming from the linear actuator.

**Hardware Interrupt**

A hardware interrupt is an electronic signal sent to the processor from an external device. Such a signal can come from a disk controller or an external peripheral. For example, when we press a key on the keyboard or move the mouse. These actions trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

**Software Interrupt**

A software Interrupt is caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is caused by the processor. For example, if the processor’s arithmetic logic unit runs a command to zero, a divide-by-zero exception is triggered, thus causing the computer to abandon the calculation or display an error message.

When an interrupt occurs, the microcontroller runs the interrupt service routine. The runtime of an interrupt would look like this:

Diagram

Description automatically generated

Figure 11: Interrupt Service Routine runtime

Figure 11 depicts how an ISR affects the main code runtime. Where the up arrows indicate the interrupt being triggered and the down arrows indicate the end of the interrupt service routine. Interrupts can be configured to be triggered by a method called signal edge detection. Signal edge is defined as the transition of a signal from a high state (3.3V on an Arduino) to a low state (0V) or vice-versa. Depending on the type of transition, three are three different types of edge detection:

**Riding edge:** when the input signal is transitioning from a low state to a high state

**Falling edge:** when the input signal is transitioning from a high state to a low state

**Either edge:** when the input signal is changing state, from high to low or low to high

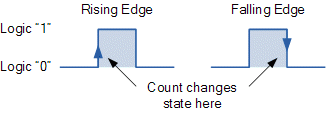


Figure 12: Rising and Falling edge graphic representation

**Pulse Width Modulation Fundamentals**

Pulse Width Modulation (PWM) is a way to describe a specific type of digital signal (a signal with a finite number of values i.e., 0V to 3.3V). PWM can generate analog signals by using a digital source. The way this is performed is via a repeating series of pulses. These pulses occur at a steady rate, but the length of each pulse varies depending on the simulated voltage magnitude. We refer to this variance in pulse length as the *Duty Cycle* of a PWM wave, which we can represent as a percentage using the following equation:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

Where the D is the Duty Cycle, the Pulse width is measured in seconds and the Frequency is in Hz. E.g., if the pulse width is 0.0005 sec and the frequency is 1000Hz, the Duty Cycle will be 50%, simulating half of the original input voltage level.

A screenshot of a computer

Description automatically generated with low confidence

Figure 13: 25%, 50%, and 75% Duty Cycle Examples

Once you know the Duty Cycle percentage and the input voltage you can easily calculate the simulating output voltage of the PWM pulse with

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

Let’s take for example the first pulse of the image above: The Duty Cycle is 25% and the input voltage is 3.3V (the pulse goes from 3.3V to 0V and vice-versa), therefore the output voltage of this PWM is 25% of 3.3V or 0.825V.

It is worth noting that this value is not a *true* analog value, but rather an average of the DC input voltage. Therefore, to read such values, there needs to be proper circuitry or code in place to be able to read a PWM.

Chart, bar chart

Description automatically generated

Figure 14: 100 kHz PWM wave with 50% Duty Cycle

The pulse above has a width of 5µs and a period of 10µs. Thus, the duty cycle is 50% and the PWM fundamental frequency is 100 kHz. Note also that the amplitude of the signal is 3.3V.

**Sending a PWM with Arduino**

The Arduino IDE has a built-in function *analogWrite()* which can be used to generate a PWM signal. The frequency of this generated signal for most pins will be about 1000Hz (Arduino Due) and you can give the function a value from 0-255.

analogWrite(0); means a signal of 0% duty cycle.

analogWrite(127); means a signal of 50% duty cycle.

analogWrite(255); means a signal of 100% duty cycle.

**Receiving a PWM with Arduino**

Unfortunately, the Arduino due or any other Arduino board, as far as I know, does not have the ability to read a PWM signal on its own. If you were to read a PWM you need to add the PWM smoothing hardware and feed this output onto one of the ADC pins on the Arduino.

Although this method yields good results, there is a way to avoid adding extra hardware to your system by adding a code workaround to the Arduino file using interrupts. The *LinearActuatorControlSystem.ino* is a good example of this technique: the code is receiving two different PWM signals from the linear actuator I/O pins. The two signals represent the current speed and position of the linear actuator respectively. The code is set up in such a way that each time the two analog pins change their state from *HIGH* to *LOW* or *LOW* to *HIGH* an interrupt will be triggered.

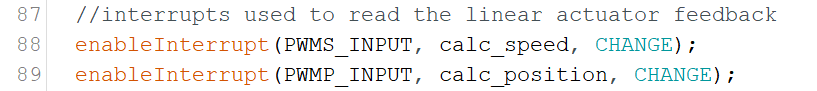


Figure 15: Interrupt setup example

In Figure 16 we can observe three PWM waves with different duty cycles. The downward arrows represent interrupts that have been triggered on a falling edge and the upward arrows represent interrupts that have been triggered on a rising edge.

A picture containing text

Description automatically generated

Figure 16: Visual representation of when an interrupt is trigged

By incorporating signal edge detection inside the ISR of the pin that has triggered the interrupt we can calculate the duty cycle of the incoming signal. This process is simple: let’s say an interrupt is attached to pin A1, and said pin is receiving a PWM pulse. This interrupt is configured to be triggered any time there is a change of state, meaning that each time A1 pin goes from high to low or low to high, an interrupt will be triggered, and the processor will leave the main code and execute an IRS (See picture 11). The IRS will then perform a signal edge detection to figure out what type of edge has triggered the interrupt. If the trigger was caused by a rising edge means that we are at the beginning of the PWM pulse. The IRS then records the time the rising edge interrupt has been triggered. Naturally, the next executed interrupt will be the falling edge, the IRS also records the time stamp of this interrupt, once we have the time stamps of the rising and falling edge interrupts, we can subtract the falling edge from the rising edge time stamp, and we obtain the period in which the PWM wave has been high (logic 1).

A picture containing chart

Description automatically generated

Figure 17: Signal edge detection

|  |  |  |
| --- | --- | --- |
|  |  | (3) |
|  |  |  |

This value can then be used to calculate the duty cycle or the intended output voltage of said PWM. Let’s take for example the PWM signals we get from the linear actuators: they are characterized as 1kHz, 5V PWM wave. The frequency of this pulse tells us how frequently a pulse will end and a new one will start.

|  |  |  |
| --- | --- | --- |
|  |  | (4) |

Where *T* is the period of the wave and *f* is the frequency. This means that with a wave of 1kHz, a new wave will start every 0.001 seconds. Let’s pretend that the time stamps of our rising and falling edge interrupts are 0.1005sec and 0.1010sec respectively, using equation () we have Δ*t* = 0.0005sec. if we take the percentage with respect to the period of the PWM we get:

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

This means that the PWM wave has been high for 50% and low 50% of the time. Finally, we can use equation (2) to determine Vout since we already know Vin.

In summary, if the PWM duty cycle were to change, the time stamps would also change, and we use interrupts and signal edge detection to collect said time stamps, compare them with each other, and calculate the duty cycle of the incoming signal.

**Voltage Divider**

It’s important to note that the rated voltage of the linear actuator PWM is 5V, and the rated voltage of the Arduino Due pins is 3.3V. This higher voltage can damage the board, so you need a way to step down this voltage. An easy fix is to use a voltage divider between the linear actuator output and the Arduino input pin.

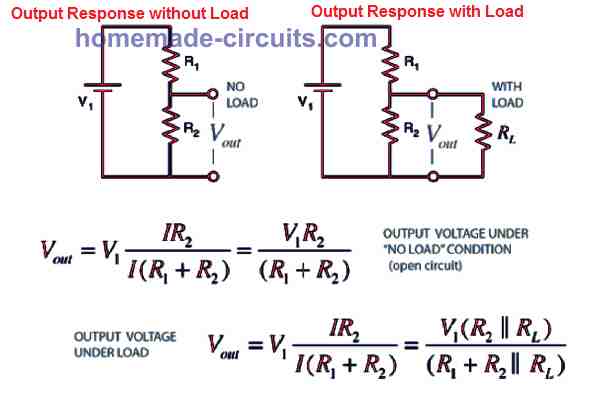


Figure 18: Voltage divider circuit

**PID Fundamentals**

Proportional-Integral-Derivative (PID) control is a common control algorithm widely used due to its robust performance in a vast range of operating conditions and partly to its functional simplicity. The PID algorithm consists of three basic coefficients: proportional, integral, and derivative which are varied to get an optimal response. We use two different PID controllers to control different behaviors from the linear actuator. The first one controls the speed of the linear actuator and the second controls its position/actuator stroke.

The basic idea behind a PID controller is to read a sensor (feedback), then compute the desired actuator output by calculating proportional, integral, and derivative responses and summing those three components to compute the output.

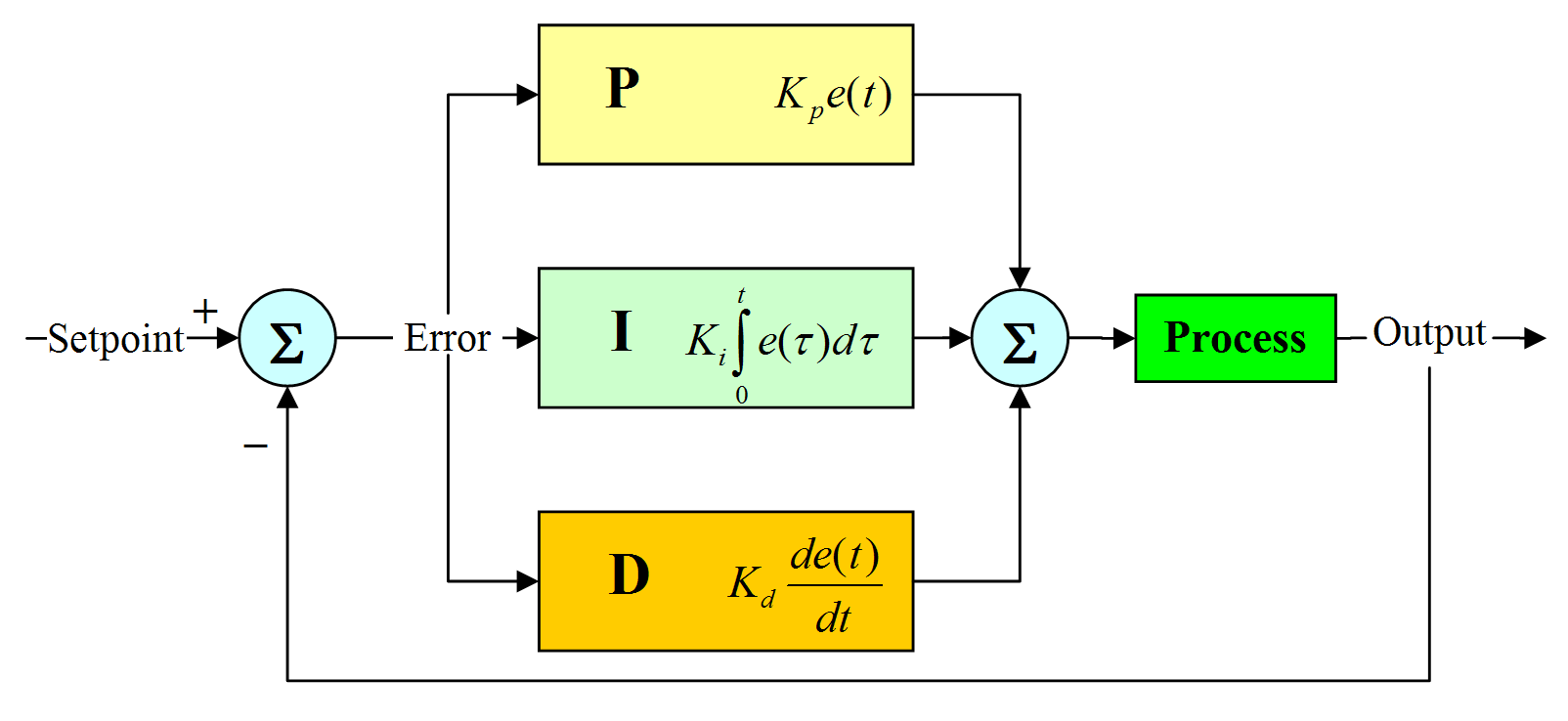


Figure 19: PID control block diagram

**Closed Loop System**

In a typical control system, the *process variable* is the system parameter that needs to be controlled, such as temperature (ºC), pressure (psi), or in our case actuator speed and distance. A sensor is used to measure the process variable and provide feedback to the control system. The *set point* is the desired or command value for the process variable, such as 7 inches in the case of the toStanding() function of the main code in our suit. At any given moment, the difference between the process variable and the set point is used by the control system algorithm (compensator), to determine the desired actuator output to drive the system (plant). For instance, if the measured stroke length of the linear actuator stroke is 5in and the desired length setpoint is 7in, then the actuator output specified by the control algorithm might be to drive the actuator forward. Driving the actuator forward causes the system to become closer to the setpoint. This is called a closed loop control system, because the process of reading sensors to provide constant feedback and calculation for the desired actuator output is repeated continuously and at a fixed loop rate as illustrated in figure 20.

Diagram, shape

Description automatically generated

Figure 20: Block diagram of a typical closed loop system.

**Proportional Response**

The proportional component depends only on the difference between the set point and the process variable. This difference is referred to as the *Error* term. The *proportional gain (Kc)* determines the ratio of output response to the error signal. For instance, if the error term has a magnitude of 10, a proportional gain of 5 would produce a proportional response of 50. In general, increasing the proportional gain will increase the speed of the control system response. However, if the proportional gain is too large, the process variable will begin to oscillate as seen in Figure 21 below. If Kc is increased further, the oscillations will become larger, and the system will become unstable and may even oscillate out of control.

**Integral Response**

The integral component sums the error term over time. The result is that even a small error term will cause the integral component to increase slowly. The integral response will continually increase over time unless the error is zero, so the effect is to drive the Steady-State error to zero. Steady-State error is the final difference between the process variable and set point.

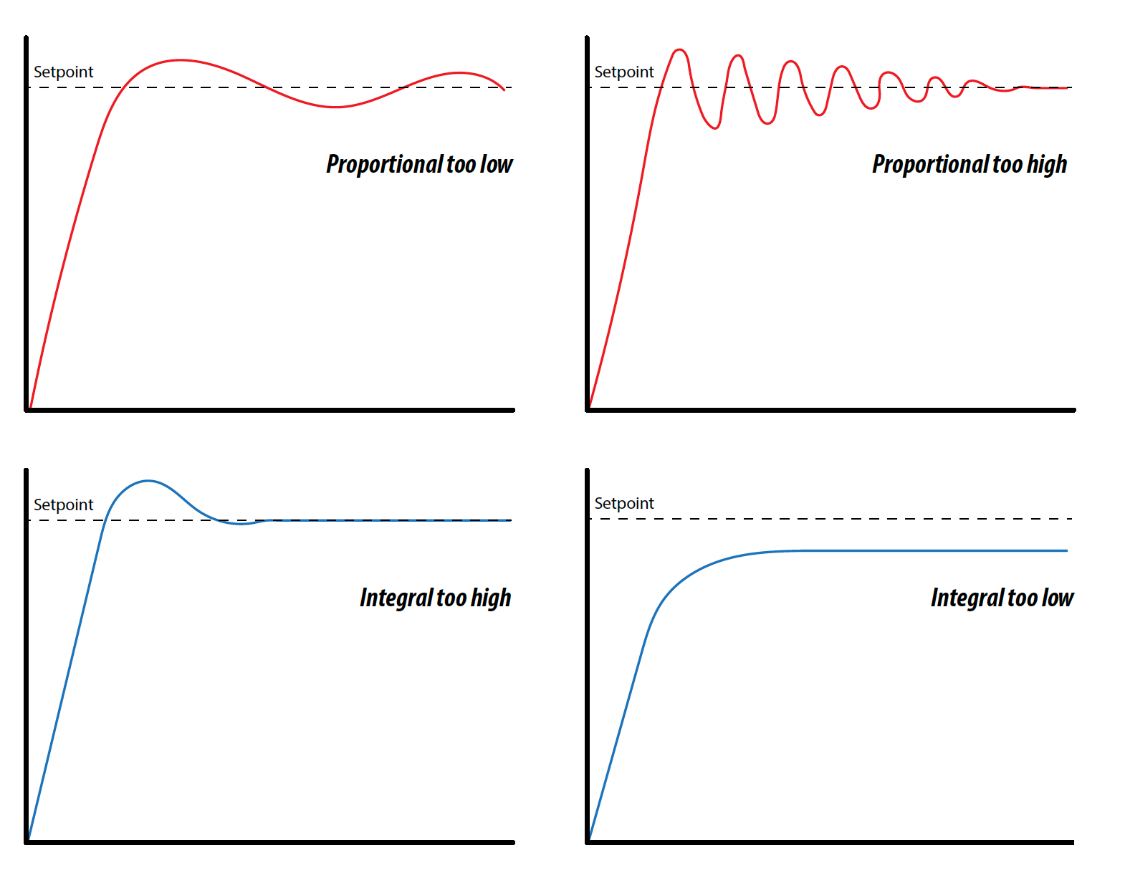


Figure 21: Example graphs of PID controller parameters

**Derivative Response**

The derivative component causes the output to decrease if the process variable is increasing rapidly. The derivative response is proportional to the rate of change of the process variable. Increasing the derivative constant *Kd* will cause the control system to react more strongly to changes in the error term and will increase the speed of the overall control system response. Most practical control systems use a very small derivative constant because the Derivative Response is highly sensitive to noise in the process variable signal. If the sensor feedback signal is noisy or if the control loop rate is too slow, the derivative response can make the control system unstable.

Chart, line chart

Description automatically generated

Figure 22: Control signals for P and PI, and for PD and PID control

It is worth noting that we don’t use derivative control on the suit since it tends to overshoot the Setpoint and oscillate until the error is driven to 0. The current suit control system uses PI control to drive the actuator position and velocity to their respective setpoints.

**Accelerometer/Gyroscope Fundamentals**

Electronic Accelerometers and/or Gyroscopes are two types of sensors that make use of inertia to obtain their readings. They can measure either linear acceleration (accelerometer) along one or several axes or angular motion (gyroscope) about one or several axes. It is worth noting that we use the *MPU6050* chip on the suit which has an accelerometer, gyroscope, and temperature sensor integrated on the same chip.

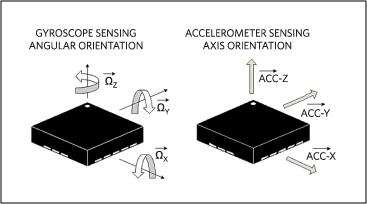


Figure 23: Angular vs linear motion

**Communication Protocol Fundamentals**

All information and communications technology rely on standardized communications protocols to operate effectively. A communication protocol is a set of formal rules describing how to transmit or exchange data. There are different types of data transfer available in digital electronics such as serial communication and parallel communication. Since parallel communication requires more wires to set up, it is not a good communication protocol for our purposes. Hence this document will not dive into parallel communication protocols.

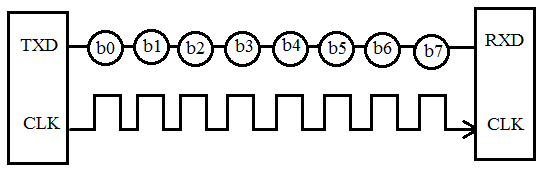


Figure 24: Serial Communication Protocol

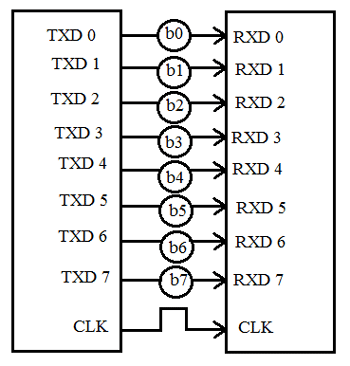


Figure 25: Parallel Communication protocol

**Serial Communication Protocols**

Serial communication is the most widely used approach to transfer information between data processing peripherals. Every electronics device whether it is a personal computer or phone runs on serial communication. This protocol is a secure and reliable form of communication having a set of rules addressed by the source host (sender) and destination host (receiver).

**Transmission modes in Serial Communication**

The serial communication data is sent in the form of bits i.e., binary pulses. Where a binary “1” represents a logic HIGH and binary “0” represents a logic LOW. Three are several types of serial communication protocols depending on the type of transmission mode and data transfer. The transmission modes are classified as Simplex, Half Duplex, and Full Duplex.

**Simplex Method**

This method is a one-way communication technique, so if the sender is transmitting the data, then the receiver can only accept data and vice versa. Some examples of the simplex method are Television and Radio.

**Half Duplex Method**

In half duplex, both sender and receiver can be active but not at the same time. So if the sender is transmitting then the receiver can only accept the incoming data and vice versa. An example of the half duplex method is the internet where the user sends a request for data, and it gets it from the server.

**Full Duplex Method**

In full duplex, both receiver and transmitter can send data to each other at the same time. A good example of this is a mobile phone.

**Synchronous Serial Interface**

The clock is also an important characteristic of serial communications, malfunction of the clock results in unexpected data transmission or even data loss. So, synchronizing the clock becomes very important.

In this type of interface, all the devices use a single CPU bus to share data and clock. The data transmission becomes faster with the same bus to share clock and data. Also, there is no mismatch in the baud rate in this interface. The well-known examples are I2C and SPI communication protocols (we use I2C to receive data from the gyroscopes).

**I2C Serial Communication**

Inter-Integrated Circuit (I2C) is a two-line communication between different ICs or modules. The two lines are called SDA (Serial Data Line) and SCL (Serial Clock Line). Both the lines must be connected to a positive supply using a pull-up resistor. I2C can deliver speeds up to 400Kbps and it used a 10-bit or 7-bit addressing system to target a specific device on the i2c bus/ This means that it can connect up to 1024 devices with a unique address. The communication length is limited so this method is commonly used to onboard communication.

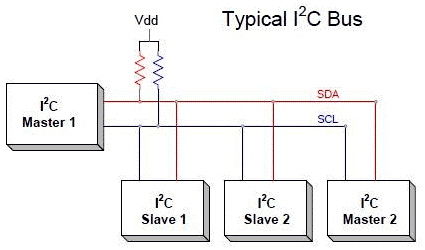


Figure 26: I2C schematic setup

At any given time, only the master will be able to initiate the communication. Since there is more than one slave on the bus, the master must refer to each slave using a different address. When addressed only the slave with that address will reply back with the information requested.

There is some set of conditions that frame a transaction. Initialization of transmission begins with a falling edge of SDA, which is defined as the ‘START’ condition. Meaning that the start of the message will be indicated when SCL is high while SDA is set to low. Subsequently, a transmission ends at the rising edge of SDA.

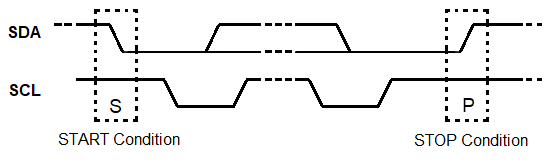


Figure 27: START and STOP conditions of a transmission

When the START condition is triggered all the devices on the same bus go into listening mode. When a transmission is started, and all devices are listening the master transmits the address of the device the master wants to communicate to.

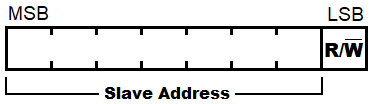


Figure 28: Address transmission format

Where MSB means Most Significant Bit and LSB means Least Significant Bit. R/W bit indicates the direction of transmission of the following bytes, if it is HIGH means the slave will transmit, and if is LOW means the master will transmit.

Each bit is transmitted on each clock cycle, so it takes 8 clock cycles to transmit a byte. After each byte is either sent or received, the ninth clock cycle is held for the ACK/NACK (acknowledged/not acknowledged). This ACK bit is generated by either slave or master depending upon the situation. For ACK bit, SDA is set to low by master or slave at the 9th clock cycle. So, if it is low it is considered as ACK otherwise NACK.

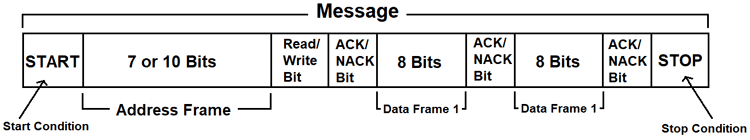


Figure 29: I2C Message frame

**Asynchronous Serial Interface**

The asynchronous type of serial protocol is very essential when it comes to long-distance reliable data transfer. Asynchronous communication does not require a timing clock as the name suggests, and as usual, high voltage means logic 1 and low voltage means logic 0.

**UART**

URAT stands for universal asynchronous receiver/ transmitter. UART is very simple and only uses two wires between transmitter and receiver to transmit and receive in both directions. Both ends also have a ground connection. UART communication can be simpex, half-duplex, or full-duplex.

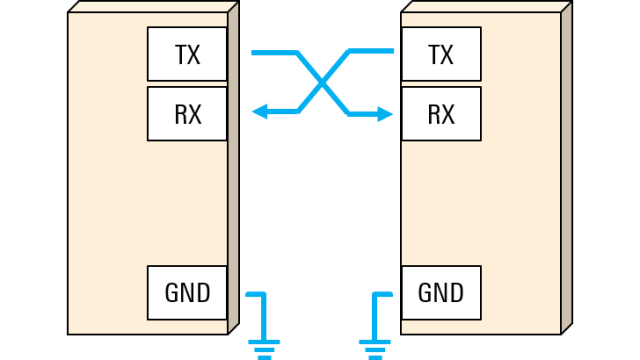


Figure 30: UART communication setup

Since the two devices that are communicating do not share a clock, both ends must transmit at the same, pre-arranged speed to have the same bit timing. The most common UART baud rates in use are 4800, 9600, 19.2k, 57.6k, and 115.2k.

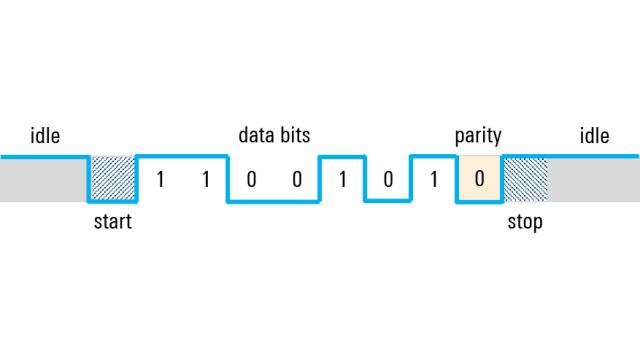


Figure 31: UART Data frame

Because UART is asynchronous, the transmitter needs to signal that data bits are coming. This is accomplished by using the start bit. The start bit is a transition from the idle high state to a low state and is immediately followed by user data bits. After the date buts are finished, the stop bit indicates the end of user data. The stop bit is either a transition back to the high or idle state or remaining at the high state for an additional bit of time.

The data bits are the user data or “useful” bits and come immediately after the start bit. There can be 5 to 9 user data bits, although 7 or 8 bits are most common. These data bits are usually transmitted with the least significant bit first. For example: if we want to send the capital letter :S: in 7-bit ASCII, the bit sequence is 1010011. We first reverse the order of the bits to put them in the least significant bit order, that is 1100101, before sending them out. After the last data bit is sent, the stop bit is used to end the frame and the line returns to the idle state.

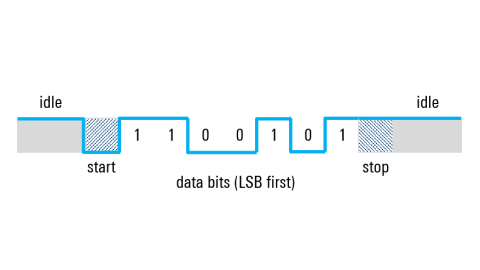


Figure 32: UART Data Transmission

**References**

<https://www.arduino.cc/en/Guide/Introduction>

<https://store.arduino.cc/products/arduino-due>

<https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm#:~:text=An%20interrupt%20is%20a%20signal,(ISR)%20or%20Interrupt%20Handler>.

<https://www.tutorialspoint.com/arduino/arduino_overview.htm>

<https://how2electronics.com/measure-tilt-angle-mpu6050-arduino/>

<https://www.maximintegrated.com/en/design/technical-documents/app-notes/5/5>

<http://hyperphysics.phy-astr.gsu.edu/hbase/electric/voldiv.html>

<https://circuitdigest.com/tutorial/serial-communication-protocols>