Foreword

Welcome to this lab series!! This lab guide is intended to be a hands-on manual for learning how to use the CH32V microcontrollers in your embedded projects. The idea was that persons who wish to move on from the Arduino for whatever reason, will be able to use this guide as a quick way to move to a cheap and popular RISC-V platform with the CH32V. This guide seeks to teach you "just enough" where you will be able to do quick projects you would usually use an Arduino UNO for. The focus on this lab guide is on peripherals, which of course is where you spend most of your time in embedded systems. You would usually get a sensor or board and then use a particular sensor on your microcontroller to read it. This book intends to teach you how to do that, which is what you would use low-cost microcontrollers for most of the time anyway. The idea is that you can already use functions like digitalRead() or analogWrite(), so it's just a matter of showing you to do what you already know to do on a new RISC-V platform.

This lab guide uses the CH32V003F4P6 device from WCH, Nanjing Qinheng Microelectronics Co. Ltd. Why? Well, it's widely available from both eastern and western suppliers, it's cheap and is the infamous "$0.10" 32-bit RISC-V microcontroller. Once you get a hang of this device, you'll have no problem using other devices in the family. This lab guide is not exhaustive, but provides a good path to learning about how to work with the CH32V assuming you have some previous experience using Arduino devices. I tried my best to use minimal specialized hardware and instead focus on using "jellybean" devices that anyone, anywhere in the world should have access to, so that you would be able to follow along with the lab guide very easily.

Armstrong Subero

Electronics Engineering Academy

Acknowledgement

Trademark Notice

RISC-V® is a registered trademark of RISC-V International. This material is not affiliated with or endorsed by RISC-V International.

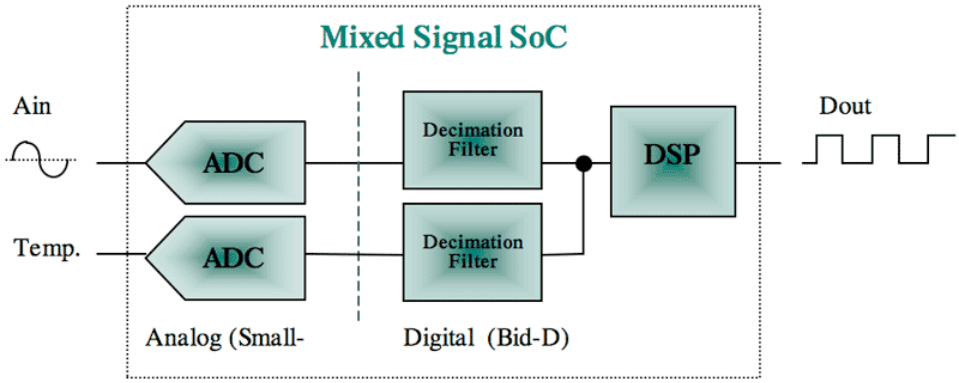
CH32 and WCH are trademarks of Nanjing Qinheng Microelectronics. This material is not affiliated with or endorsed by WCH.

AVR® is a registered trademark of Microchip Technology Inc. This material is not affiliated with or endorsed by Microchip Technology Inc.

Arduino® is a registered trademark of Arduino SA. This material is not affiliated with or endorsed by Arduino.

Version: 1.0

Mixed Signal Embedded Systems



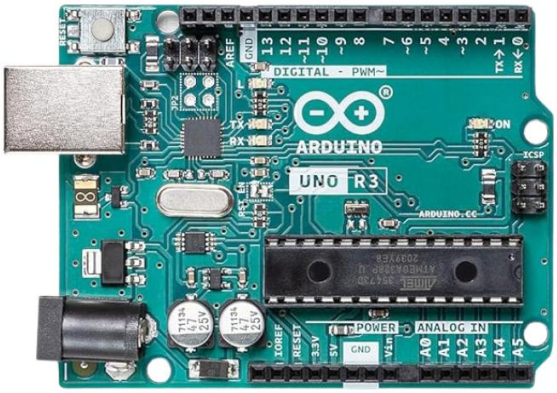
Welcome to this lab series!! In this guide I'll teach you about mixed signal embedded systems using CH32V RISC-V microcontrollers. An embedded system has one job. To do a specific job and to do it reliably. These systems aren't meant to be flexible. They're meant to run a single program and keep it running for decades and do so reliably and usually while running on quiescent current or a small battery. Sure, we have embedded systems these days that integrate Bluetooth and WiFi and all those stuffs, but as a former Arduino user you know that not every system needs to be connected to the internet!

At a basic level, every embedded system is built from three things, there is a processor that executes instructions and moves data around. There are peripherals that handle timing, communication and interaction with external signals and there is firmware that ties everything together and defines how the system behaves. If we change anyone of these our system will change, sometimes in very subtle ways that are not immediately obvious. The processor is the part most people think about first when they think about embedded systems, but more often than not, it is not the most important part. Many tasks are handled by peripherals because hardware can do them more reliably and with better timing than software loops ever could.

That's where mixed signal embedded systems come in. If you come from an Arduino background, you may be used to thinking about software first, and treating hardware as an afterthought, just a module or device to connect to. As you work through these labs, I hope you can appreciate the hardware at a deeper level.

That's not to say software is not important. In your embedded system, firmware is not an application layered on top of the system. Firmware IS the system. When power is applied, the firmware starts executing and never really stops. There is no clean exit point, no shutdown sequence most of the time and no operating system to hide the mistakes. If the firmware misconfigures a peripheral or mishandles time, the hardware will still do exactly what it was told to do. Speaking of time, one of the biggest differences between embedded system and general-purpose computers is the role of time. In embedded work, timing is not an optimization detail. It is a design constraint. Signals must be sampled at the right moment. Outputs must change at predictable intervals. Events must be handled quickly enough that the outside world does not notice failure. Ignoring timing is one of the fastest ways to build a system that works on the bench and fails in the field. Coming from an Arduino background you will appreciate this fact; you would have used things like millis() and delays. Which brings us to our other section. Why it's time to leave the Arduino behind.

Leaving Arduino Behind



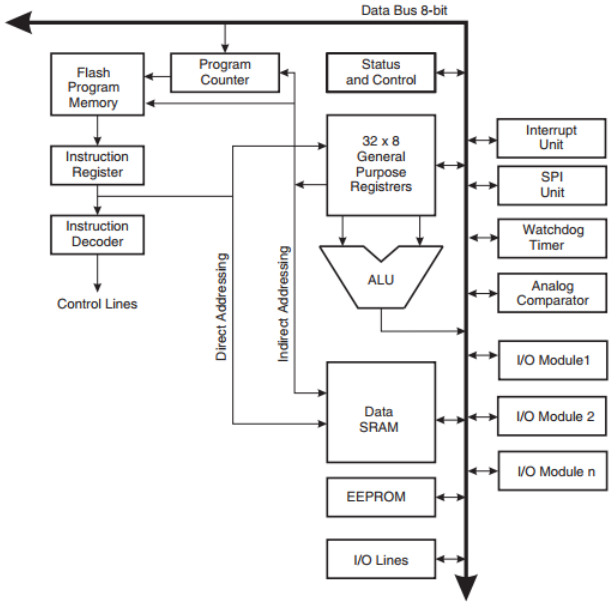
The Faithful Arduino Uno R3

So, as we were saying in the last section, embedded systems are defined by time. Because of this, many embedded systems are structured around a simple repeating loop. The program initializes the hardware and then continuously checks inputs, updates internal state, and drives outputs. This structure is often called a super loop. It looks primitive, but it is remarkably effective and still forms the backbone of many commercial products. More complex systems often grow out of this pattern rather than replacing it entirely.

Platforms like Arduino deliberately hide much of this reality. They take care of startup code, clock configuration, peripheral setup, and timing details behind simple function calls. This lowers the barrier to entry and makes experimentation easy. The downside is that it also hides how the system actually works. Many people reach a point where they can make things function but cannot explain *why* they function, or why they sometimes fail. This is where studying embedded digital systems at a lower level becomes valuable. Architectures like RISC-V make it easier to see what is going on under the hood. Registers are visible which means initialization is explicit. Peripherals behave exactly as described in the datasheet, with no magic in between. Working at this level forces you to understand the system instead of memorizing library calls.

The goal here is not to abandon high-level tools or nostalgia for older platforms. The goal is to build a mental model of how embedded systems actually operate. Once that model exists, the choice of microcontroller matters much less. GPIO is still GPIO. Timers still count and communication buses still move bits according to strict rules. Those fundamentals do not change, even when the tools do. In fact, you already know a lot about how embedded systems operate under the hood! I'm counting on this knowledge you gained to teach you about how CH32V microcontrollers work. Understanding embedded digital systems is about learning to think in terms of signals, time, and state. It is about designing behavior rather than writing code that happens to work. Everything that follows in this book builds on that foundation. The other major reason you should move away is cost!! An Arduino UNO R3 costs between US $20 and US $30!! A CH32V003F4P6 device can be bought for as little as $0.10.

Recap of AVR MCU



The AVR MCU

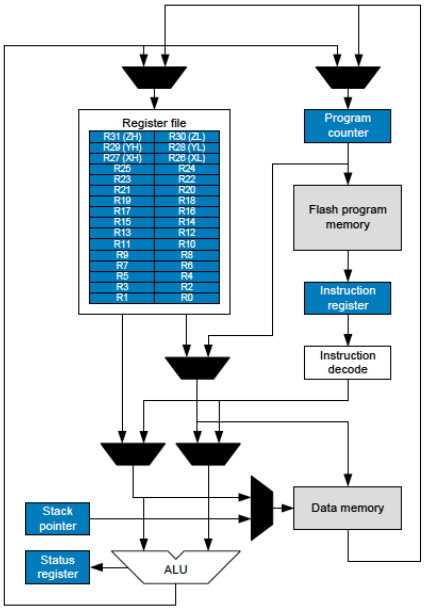
As a quick recap, the Arduino UNO is built around the ATmega328P, an 8-bit AVR microcontroller that many people unknowingly used as their first real embedded CPU. The AVR core was designed to be simple, predictable, and efficient, with a strong focus on deterministic behavior. It uses a Harvard architecture, separating program memory from data memory, which allows instructions to be fetched while data is accessed. Most instructions execute in a single clock cycle, making timing relatively easy to reason about compared to more complex processors.

One defining feature of the AVR core is its heavy use of registers. The CPU provides thirty-two general-purpose 8-bit registers that are directly connected to the ALU. This allows many operations to be performed without touching memory, which is important in a system with limited SRAM and modest clock speeds. Peripheral control is handled through memory-mapped registers, meaning GPIO, timers, serial interfaces, and interrupts are all configured by setting and clearing bits in control registers, even if Arduino normally hides this behind library calls.

The interrupt system on the AVR is straightforward and fixed in priority, which limits flexibility but makes system behavior predictable. Combined with the super loop structure commonly used in Arduino sketches, this simplicity is one reason the platform feels stable and forgiving for beginners. At the same time, the 8-bit nature of the core imposes real limits. Larger data types, higher precision math, and more demanding peripherals quickly expose the constraints of the architecture.

The AVR core is very powerful though, and ingenious persons have shown through the Arduino platform what's possible with a simple 8-bit MCU. However, once you move on from the AVR to RISC-V, a whole new world of possibilities opens up to you.

The AVR CPU Core



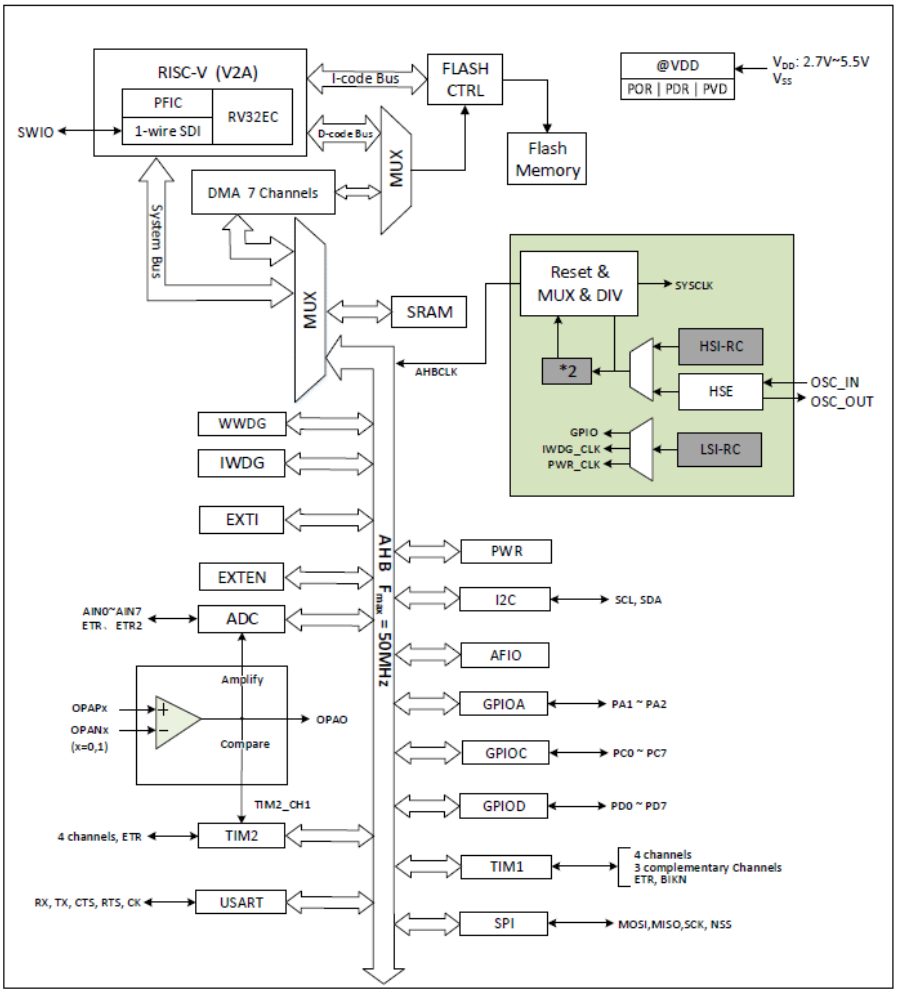
The AVR CPU is the Core of the MCU

This diagram shows the internal data flow of the AVR core used in the Arduino UNO and helps explain why it feels simple and predictable in practice. At the center is the register file, which contains thirty-two general-purpose 8-bit registers directly connected to the ALU. Because operands can be taken straight from registers and results written back without going through memory, many instructions complete in a single clock cycle. This tight coupling between registers and the ALU is one of the defining characteristics of the AVR architecture and is a major reason it performs well despite being an 8-bit CPU.

Instruction flow begins at the program counter, which holds the address of the next instruction in Flash program memory. That instruction is fetched into the instruction register and passed to the instruction decode logic, which determines what operation should occur and which registers or peripherals are involved. Because program memory and data memory are separate, instruction fetch can occur independently of data access. This separation simplifies timing and makes execution behavior easier to predict, which is especially important in small real-time systems.

Data movement and system state are handled through the ALU, data memory, stack pointer, and status register. The ALU performs arithmetic and logic operations and updates the status register flags, which influence conditional branches and interrupts. Data memory is used for SRAM variables, while the stack pointer manages function calls, local variables, and interrupt context saving. Together, these blocks form a straightforward execution model: fetch an instruction, operate on registers, optionally access memory, update flags, and repeat. This clarity is why the AVR core is such a good reference point when transitioning to more modern architectures like RISC-V.

The CH32V MCU



The CH32V MCU

The CH32V MCU we will be using is not much different from the AVR MCU. We have our RISC-V core, and associated peripherals that it can connect to. If you used other microcontrollers before, you will be familiar with the AHB (Advanced High-performance Bus) typically found in ARM microcontrollers. In case you aren't familiar however its purpose is to act as the high-speed backbone bus that moves large amounts of data efficiently between the CPU and peripherals. The device has the register interface you are familiar with from the Arduino and you get all the peripherals you are accustomed to SPI, UART, I2C, PWM ADC etc. however they do feel more advanced than the AVR versions of the peripherals.

Despite being more advanced however, these are very easy to use, and the software handles a lot of the abstraction for you so you'll feel right at home. The device we use in this lab guide in particular is nearly 1:1 compatible with the 8-bit AVR MCU in terms of peripherals and it feels like a good starting point to replace an 8-bit AVR device. But before we get to peripherals, let's talk a little bit about the RISC-V core.

Understanding RISC-V CPUs

Looking at the diagram in the last section, if we zone in on the RISC-V core entitled "RISC-V (V2A)" you will observe that in that block, the RISC-V CPU the device is using is the "RV32EC" device. This core like any other RISC-V core has a designation. This is because RISC-V has base instructions and extensions that support a range of processors from deeply embedded to cloud data center applications.

RISC-V has 32- 64 and 128-bit base instructions which are designated RV32I, RV64I and RV128I respectively. The 32-bit instruction set additionally has a subset with only 16 registers designed for deeply embedded processors called the RV32E. A processor designed around RISC-V must support the base architecture, but doesn't need to support any extensions. The common extensions are:

Extension	What It Does
M	Integer multiply and divide
F	Single-precision Floating Point
D	Double-prevision Floating Point
Q	Quad-precision Floating Point
C	Compressed Instructions
A	Atomic Instructions
B	Bit Manipulation
L	Decimal Floating Point
J	Dynamic Translated Languages
N	User Level Interrupts
T	Transactional Memory
P	Packed-SIMD Instructions
K	Extension for Scalar Cryptography
V	Vector Operations
H	Hypervisor

S	Supervisor-Level Instructions
G	A general-purpose ISA
Zicsr	CSR Access
Zba	Address Generation
Zbb	Basic Bit Manipulation
Zbc	Carry-Less Multiplication
Zbs	Single-bit Manipulation
Zbkb	Basic Bit Manipulation for scalar cryptography
Zcb	Basic additional compressed instructions

You also have the Zcmp that supports push and pop instructions. It's just important that you understand the basics of these suffixes just to understand what the RISC-V core supports. You may for example see a core with "RV32IMACZicsr" and know that it supports the base instruction set, integer multiply and divide, atomic instructions and compressed instructions. You can also tell that you have CSR access. That's all you need to know generally to work with RISC-V at the level we're working at, as we'll usually be working with C that abstracts this away from us.

Model	Instruction set	Hardware stack levels	Interrupt nesting levels	Vector Table Free channel	Launch width	Flowline	Vector table model	Extended instruction (XW)	Cache	Memory protected areas numbers
QingKe V2A	RV32EC	2	2	2	1	2	Address/Instruction	√	×	×
QingKe V2C	RV32EmC	2	2	2	1	2	Address/Instruction	√	×	×
QingKe V3A	RV32IMAC	2	2	4	1	3	Instruction	×	×	×
QingKe V3B	RV32IMCB	2	2	4	1	3	Address/Instruction	√	×	4
QingKe V3C	RV32IMCB	2	2	4	1	3	Address/Instruction	√	×	4
QingKe V4A	RV32IMAC	2	2	4	1	3	Address/Instruction	×	×	4
QingKe V4B	RV32IMAC	2	2	4	1	3	Address/Instruction	√	×	×
QingKe V4C	RV32IMAC	2	2	4	1	3	Address/Instruction	√	×	4
QingKe V4F	RV32IMACF	3	8	4	1	3	Address/Instruction	√	×	4
QingKe V4J	RV32IMAC	2	2	4	1	3	Address/Instruction	√	I-Cache	4
QingKe V5A	RV32IMACBF	-	8	4	2	7~9	Address/Instruction	√	I/D-Cache	8

WCH RISC-V CPU Cores

Look at our inset table showing WCH RISC-V CPU cores. Looking at it, we see that the CPU in our CH32V003F4P6 is actually the "QingKe V2A" CPU. The QingKe V2A is a 32-bit microprocessor core based on the RV32E extension. As we recall the RV32 "E" extension was designed for embedded applications in mind and has a subset of the RV32 "I" register file having only 16 registers which we can work with, we only have x0-x15.

This core in particular supports the RV32EC instruction set and there is a V2C extension in families that supports hardware multiplication. This is a subset of the M extension though, which the company calls the 'm' extension meaning the device has support for multiplication but not division. The CPU is a 2-stage instruction pipeline and as is typical in two stage pipelines, we have a fetch stage and an execute stage. In the fetch stage instructions would be fetched from memory and then in the execution state the instruction would be decoded and executed. A two-stage instruction pipeline will make the device good for embedded applications as a fewer stage instruction pipeline means the device has simpler control logic and reduced switching activity. Reduced switching activity in theory translates to lower power consumption. Efficiency obtained though instruction-level parallelism in more complex designs for certain tasks requiring more on time is a good counter argument to this, but for now, you should takeaway that it's a simple processor. The core may also support some custom instructions and a special "WFE" (Wait for Event) instruction as well. This is a custom instruction because there is no WFE instruction in the RISC-V specification.

Devices that support this instruction can set the 'WFE' by simply setting the specific location 1 of the system control register in the fast programmable interrupt controller (PFIC). Unlike the classic 4-wire JTAG debug interface in RISC-V, QingKe processors pioneered the adoption of 2-wire and even 1-wire DTM (Debug Transport Module) interfaces, enabling full processor debugging and programming with just two or even one I/O pins. This is similar to debugWIRE on AVR devices, so it's very similar to what you've grown accustomed to.

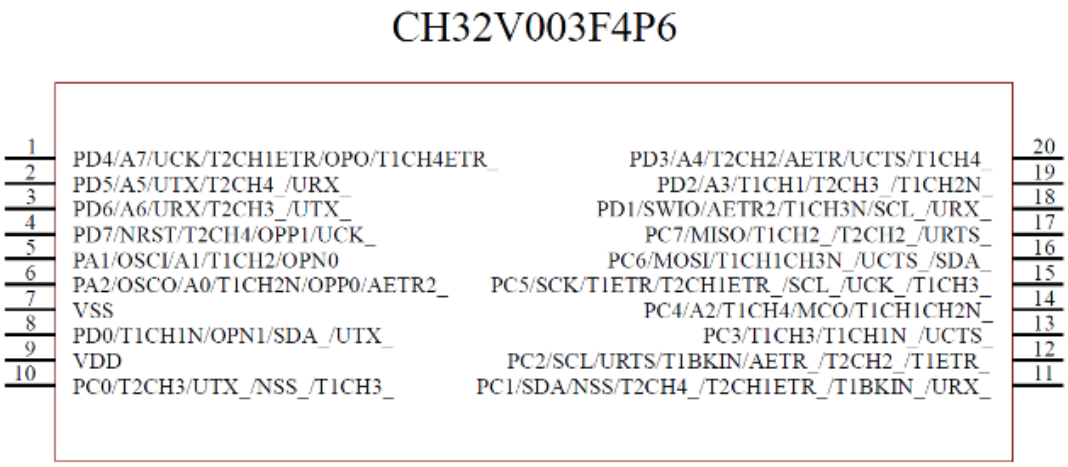
CH32V MCU Families

QingKe V2	QingKe V3	QingKe V4
CH32V003 (A)	CH32V103 (A)	CH32V203 (B)
CH641 (A)	CH573/1 (A)	CH32V208 (C)
CH32V002 (C)	CH569/5 (A)	CH32X035 (C)
CH32V006/5 (C)	CH585/4 (C)	CH32L103 (C)
CH32V007 (C)		CH592/1 (C)
		CH643 (C)
		CH32V303 (F)
		CH32V307/5 (F)
		CH32V317 (F)
		CH564 (J)

From this table we can see some of the members of the CH32V family of devices. Of these our focus will be in the CH32V003 which is part of the QingKeV2A. Besides being separated by cores. The CH32V microcontrollers can be placed into 3 distinct families based on the core they contain. Whilst all the devices are based on the QingKe cores, the QingKe cores are themselves in turn based around different cores, the QingKe, QingKe V2, the QingKe V3 and the QingKe V4 cores.

This table is by no means expansive or complete as there are many move devices in the family, but it's a good starting point to understanding where the device we are using falls in this lineup.

The CH32V003F4P6



We finally reach the point where we can look at our target device, the CH32V003F4P6. The CHV003 devices are designed for deeply embedded and industrial applications and this device is no exception. The device runs at up to 48 MHz and these microcontrollers consist of 4 parts:

- *A CPU core* – Which as we now know is an embedded specialized WCH RISC-V core. This is the brain of the microcontroller, allowing the device coordinate the execution of instruction that were written by us into the program memory.
- *Program Memory* – This holds the instructions of the device, when we write our program it is converted to a format the microcontroller and understand and is stored in program memory which is 16KB.
- *Data Memory* – This part of the CPU holds memory that we as embedded engineers use for variables, which the CPU reads from and writes to, we have 2 KB for use.
- *Pins and Peripherals* – There are special functions built into the device that perform a wide variety of functions. These special functions allow the device to be able to perform a variety of tasks.

One of the nice features of this device is its ability to run at up to 5.5v. Which means all your library of Arduino modules, parts and circuit snippets will be compatible with the device. The full operating range of the device is from 2.7v to 5.5v however it is noted in the datasheet that the device's VDD performance will deteriorate if is less than 2.9v when using the ADC. For that reason, I recommend you use the device in the 3.0v to 5.0v range. Why? Well working at above 3.0v will allow you to use the device without worry of degradation of performance and running at the full 5.5 v is pushing it to close to its limits. In practice I have run it from a 3.7 lithium-ion battery for some applications and for other applications I have run the device at 5v, the examples in this book typically assume a 5v power supply is used.

One of the important features you must consider when learning to program with a device is the flash endurance. The flash endurance of the device is the number of times you can program the device without it failing. For the CH32V003F4P6 the flash endurance is somewhere between 10 000 times and 80 000 times, and the device will retain the data you put in it for up to 10 years.

CH32V003F4P6 vs ATMEGA328P

So, we learnt a bit about the CH32V003F4P6 device and at this point you may be wondering how it stacks up to the ATMega328 in the Arduino Uno R3. Let's compare these devices starting with the device core.

Core		
Feature	CH32V003F4P6	ATmega328P
Architecture	32-bit RISC-V (RV32EC)	8-bit AVR
Core Width	32-bit	8-bit
Max Clock	48 MHz	20 MHz (16 MHz typical)

The CH32V003F4P6 has a 32-bit architecture which of course means a larger core width and runs at a faster clock speed. What about peripherals? After all these are even more important than the core for our purpose!

Hardware and Peripherals		
Peripheral	CH32V003F4P6	ATmega328P
ADC	10-bit	10-bit
UART	1	1
SPI	Yes	Yes
I2C	Yes	Yes
Timers	Advanced	Basic
Comparator	Yes	Yes
Op Amp	Yes	None
DMA	Yes	None

In terms of peripherals, the devices are matched quite well and you won't miss any peripherals moving from the Arduino to the CH32V. You also gain the power of having an Op Amp as well as direct memory access. Speaking of memory:

Memory		
Memory Type	CH32V003F4P6	ATmega328P
Flash	16KB	32KB
SRAM	2 KB	2 KB
EEPROM	None	1 KB

The devices both have the same amount of SRAM, though the ATmega328P has double the amount of flash. The CH32V003F4P6 also has no EEPROM whereas the Arduino offers 1 KB of EEPROM. While this may seem like the CH32V003 is at a disadvantage, this is where architecture makes all the difference. The CH32V003F4P6 supports compressed instructions, which means that many of these instructions are 16-bits, and we have wider registers meaning we need fewer instructions per task. In practice 16 KB of RISC-V code can feel like 24-28KB of AVR code, for many embedded tasks! Remember the Arduino uses flash for its bootloader, core and wiring framework and abstracted libraries, so a 8-10KB Arduino sketch especially math heave ones may compile to 2-4KB on the CH32V003, especially since we aren't relying on too much software overhead. 8-bit math adds multiple cycles and extra code leading to a less efficient storage structure. Code density means less flash is comparable, and it's something you'll experience as you develop your own applications. That of course leave's the development environment and experience, which is often overlooked.

Development		
Feature	CH32V003F4P6	ATmega328P
Voltage	2.7 – 5.5v	1.8-5.5v
Debugging	1-wire SWIO	ISP
Ecosystem	Small, growing	Massive (Arduino)
Cost	Ultra-Cheap	Cheap

The devices are very evenly matched when it comes to the development experience. While the Arduino does of course have a broader community, the overall experience of going from code to device is comparable. The nice thing about the CH32V003 though is that while it punches pound for pound with the Arduino, it does so at significantly less cost. The device makes an excellent replacement for the 8-bit Arduino device.

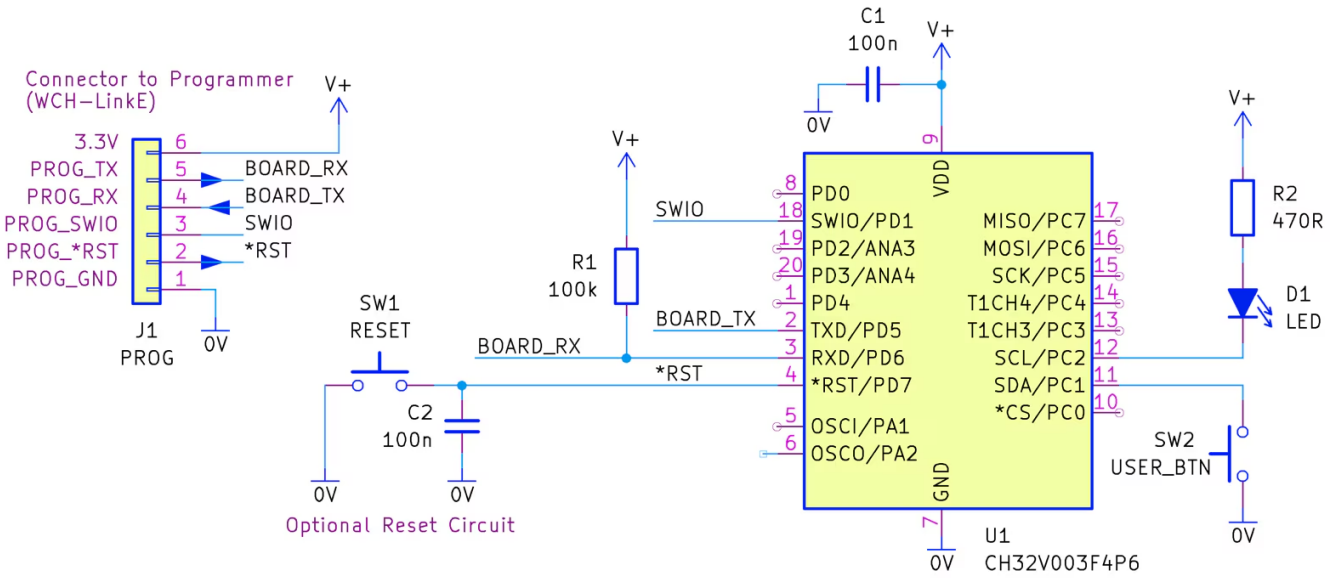
Big Change I: The External Programmer

On the Arduino device the ATMega328 has a bootloader, specifically the Optiboot bootloader which means it doesn't need to interact with any external devices to write program to the device. In order to flash the WCH CH32V microcontrollers however, the WCH-LinkE programmer/debugger is needed, though some of the larger chips can be programmed via bootloader, if you plan to do serious work with this chip, pick up a Link-E, which is easy to use as it has USB adapter and despite being low cost the design is very sturdy. Unlike other devices where you need to look at clones because the original is not easily accessible, the programmer/debugger is available at low cost directly from the manufacturer, in fact it's one of the cheapest manufacturer original debuggers you can buy. If you are buying the debugger, make sure you get the WCH-LinkE and not the WCH-Link which is an older device that will not program the RISC-V chips from the manufacturer as it is intended to program their legacy ARM line.

Before you program the device, if you connect your WCH-LinkE device and you see the blue LED lit, it means the device is in ARM mode. To get it into RISC-V mode you need to press and hold the ModeS button as you connect it to your computer. Now some of the LinkE devices come in a plastic case that has this button sealed inside. While some people pry apart the case, I personally just make a pin sized hole in the case then use that in order to push the ModeS button. This only needs to be done once, so either mechanism you use is fine. Just make sure and hold this button as you connect the LinkE programmer/debugger to your computer to get it ready to program the RISC-V devices.

This LinkE programmer is nice because we case use it to program the device via a single wire, SWIO. We still need to connect a reset to the device, but it is very handy especially when working with low-cost devices like the CH32V003F4P6. We barely lose any I/O as in theory we only need to use the SWIO pin and the reset button to program the device. The programmer also comes with built in serial communication, so if you are accustomed to on the Arduino you will have all the features you have come to expect.

Lab #1: Connecting The Programmer

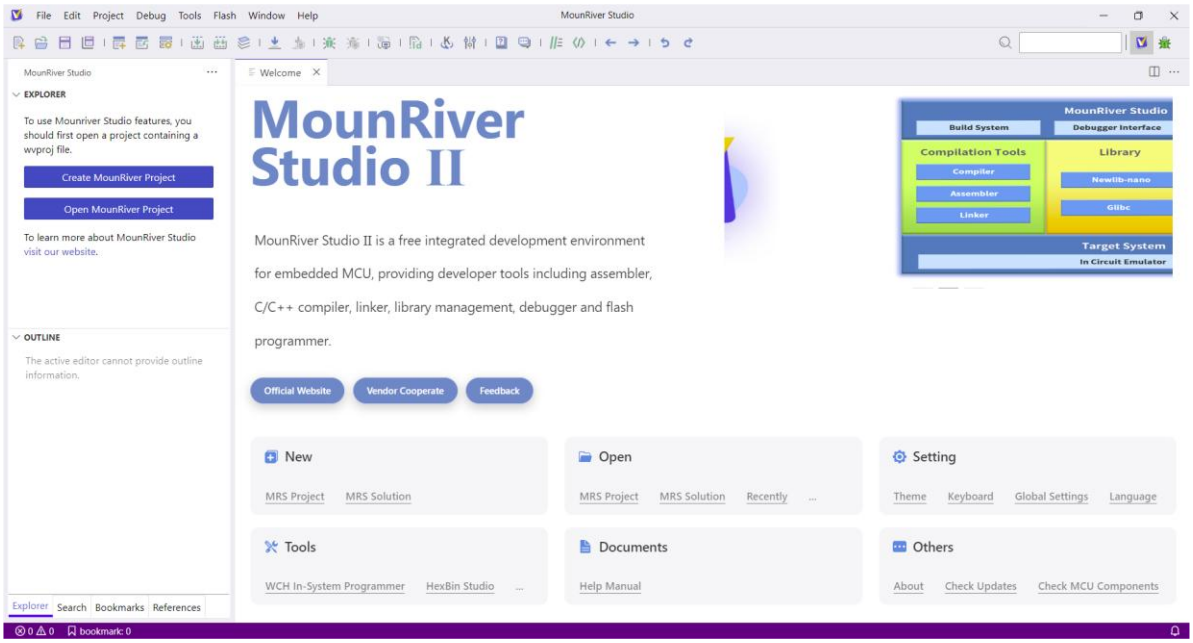


The recommend connection settings for using this device with the lab manual is this schematic. In order to connect the device, we need a power supply, our reset circuit connected and our connection to our debugger. The device can run at 5V or 3.3v, either one will be fine to follow along with the projects in this manual. It doesn't matter if our power supply is linear or switch mode as long as it can supply the 3.3v needed for the device. A power supply that can supply up to about 1A is standard for prototyping with microcontroller devices. Any jellybean 3.3v or 5v regulator is fine to use.

Though we can use an external oscillator with the device, we will use the internal oscillator in order to reduce Bill of Material (BOM) cost. The external oscillator is usually a crystal as the device has circuitry that can handle it internally. There are different development boards that handle most of these connections for you, I recommend you get one that exposes all the I/O so you won't have to worry about an I/O being occupied when you have to follow along with examples.

The development board that is sometimes bundled with the Link-E can be used, however for some of the labs you may have to modify the board since the I/O is used for other tasks, you can check the schematic diagram of the board to see if that pin is available when we need to use a certain I/O.

Big Change II: Using a Professional IDE



When prototyping with the microcontroller devices, we need to write programs in order to instruct the devices what to do. We usually use a cross-compiler that is able to compile our programs into a format the microcontroller can understand as our machine usually has a different architecture from our target microcontroller device. On the Arduino platform, this was usually handled by the Arduino IDE which is a rather basic IDE with very limited features.

We can use a variety of tools in order to program our microcontroller, however using an Integrated Development Environment (IDE) is the best option as it will simplify everything especially if you are coming from the Arduino ecosystem. When we use an IDE, we are able to very easily program our device. The IDE we will use is MounRiver Studio II which is freely available and can be used to program CH32V RISC-V microcontroller devices.

It may not seem like a big deal because in the Arduino ecosystem the IDE is free, however usually in the world of professional development it is not uncommon to have to pay hefty fees for an IDE, so this is a really welcome change. You will need to install MounRiver Studio II in order to follow along with this lab guide. You can obtain a copy of MounRiver studio from here:

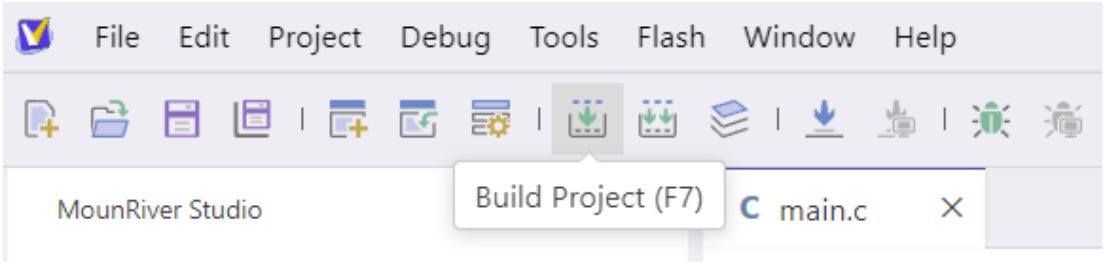
https://mounriver.com/download

Lab #2: Building Your First Project

We can use MounRiver Studio to build our application. Once you have MounRiver installed, open the CH32V003 project that's preconfigured, or start a new project. Delete the contents of main and type in the following program:

```
#include "debug.h"
int main(void)
{
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  SystemCoreClockUpdate();
  Delay_Init();

  while(1)
  {

  }
}
```

After we have typed in that program, we can build the project by clicking the build project button:



Lab #3: Examining the Build Output

After we build the program Observe the Output, look at the output window at the bottom of your screen, switch the output to "Build Output" and you will see the following output:

```
make -j16 all
make --no-print-directory main-build
riscv-none-embed-size --format=berkeley "CH32V003F4P6.elf"
   text     data      bss      dec      hex  filename
   1016       24      264     1304      518  CH32V003F4P6.elf
[21:20:32] Build Finished. (took 724ms)
------------------------------------End-------------------------------------
```

The make commands just tell the compiler to build the project using 16 parallel threads for faster compilation and we do some non-verbose printing. Pay attention to the line that says "risc-v-none-embed-size" what this is doing is telling the compiler that we want to use the size tool from the RISC-V GCC toolchain to show memory usage in Berkeley format.

The output is telling us how much memory our CH32V003F4P6.elf is using. We see:

Section	Size	Description
text	1016 bytes	Code and read-only constants (stored in flash)
data	24 bytes	Initialized global/static variables (stored in RAM + flash)
bss	264 bytes	Uninitialized global/static variables (allocated in RAM, zero-initialized at startup)
dec	1304	Total in decimal (text + data + bss)
hex	518	Total in hexadecimal (1304 in hex)

This means that our program is using a total of 1.3 KB of memory. To get the ram usage in particular, we add the .data section and the .bss section and we see the total ram used is 24 (data) + 264 (bss) or 288 bytes of ram in total. Usually after we build the program, we can download it, but since this is an empty program, even if we do download it nothing will happen.

Super Loop Architecture

The program we wrote in the last section is our base minimum program that forms part of what is known as a superloop application. You'll see our program has four "steps" each with a purpose.

Step	Purpose
NVIC_PriorityGroupConfig()	Configures interrupt priority grouping
SystemCoreClockUpdate()	Updates system clock variable
Delay_Init()	Initializes delay functions
while(1)	Runs forever (idle state currently)

A superloop is a cooperative structure we use in embedded systems, especially resource constrained devices like the CH32V003. This design typically has the following structure:

```
// 1. Includes
#include <mylibrary.h>

int main(void)
{
    // 2. Initialization
   SystemInit();
   Peripheral1_Init();
   Peripheral2_Init();
   PeripheralX_Init();

// 3. Main Loop
while(1)
{
   Read_Sensor();
   Update_Actuators();
   Handle_UART();
   Check_Timeouts();
   Blink_Led();
  }
}
```

The structure has three parts, an include section where we include our libraries, an initialization section where we configure the peripherals we want to use in the application and a main loop section where all the tasks are checked sequentially inside the program. This is done through being "polled" meaning we explicitly check for events. We say this type of design is cooperative, since each function must return quickly to avoid blocking the main loop. This makes these types of designs highly predictable as the order of execution is consistent and this type of loop is also single threaded since there is only one thread of execution. There is no kernel or real time operating system in this structure, just direct execution on the device. Now that we have our program setup, we can build the program and observe the output.

Lab #4: setup() and loop()

Now it may be difficult if you're coming from Arduino to see this, after all you're accustomed to seeing this as your bare project:

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

}

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

}
```

This is because the Arduino platform is hiding a lot from you! To make your mind at ease we can do something like this:

```
#include "ch32v00x.h"

void initMain()
{
  // put your setup code here
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  SystemCoreClockUpdate();
}

int main(void)
{
  initMain();

  while(1)
  {
    // put your main looping code here
  }
}
```

When we compile this program, it builds just like before! Your initMain (short for initialize main) is where you will put your "setup" code and you put your "looping" code inside of while (1)! That's all there is to it. This is your bare minimum setup. In main you see two function calls these are required for configuring some device internals, for now don't focus too much on what they do, just know they are needed for the device to run properly.

Notice I replaced "debug.h" with "ch32v00x.h". This results in a leaner binary size as that file just has aliases and bit definitions. In case you're curious to see them, from the IDE right click on the include file and click "Go to Definition". You'll see a file that maps everything to registers, no hidden secrets or spaghetti holding everything together, you see exactly what's going on and can easily fix it. Congratulations you've taken the first step in your journey so let's continue!!

Electronic Concept: The LED

Now that our software is setup and our microcontroller is connected, we will begin interfacing components to our microcontroller. A lot of our circuits will involve use of an LED. I'm sure the vast majority of my readers would have used an LED, however in order to cater to all level's we'll just briefly review the LED here to be safe. A Light-Emitting Diode (LED) is a semiconductor device that emits light when an electrical current flows through it. LEDs are highly efficient, durable, and versatile, making them an essential component in modern electronics. They are used in applications ranging from indicator lights and displays to high-intensity lighting and optical communication systems. You will use them in almost every project you will do with the CH32V as they are just that versatile.

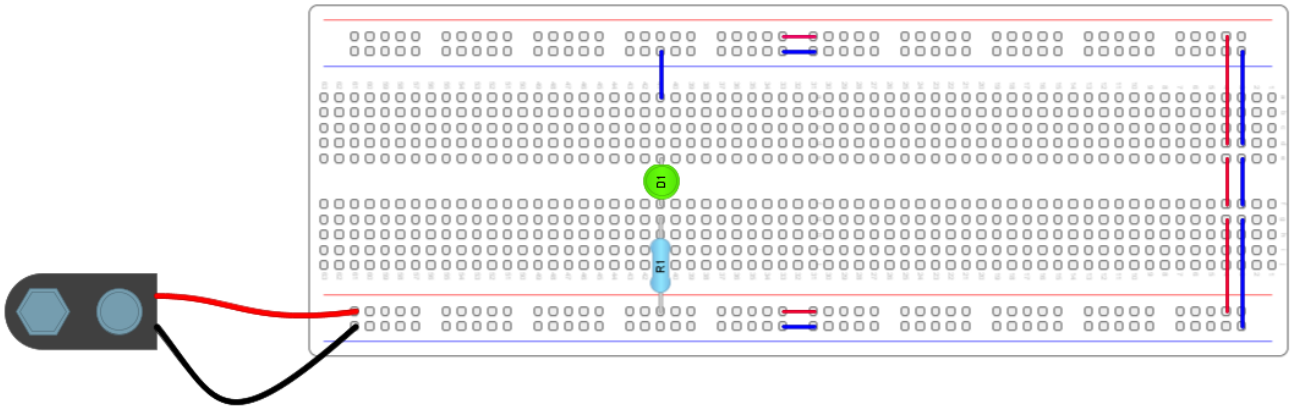
Light Emitting Diode

The operation of an LED is based on the principle of electroluminescence. When a forward voltage is applied across the LED's terminals, electrons in the semiconductor material recombine with holes at the junction of the p-type and n-type layers. This recombination releases energy in the form of photons, producing light. The wavelength (and thus the color) of the emitted light depends on the energy gap of the semiconductor material used. Common materials include gallium arsenide (GaAs) for infrared LEDs and gallium phosphide (GaP) for visible light LEDs.

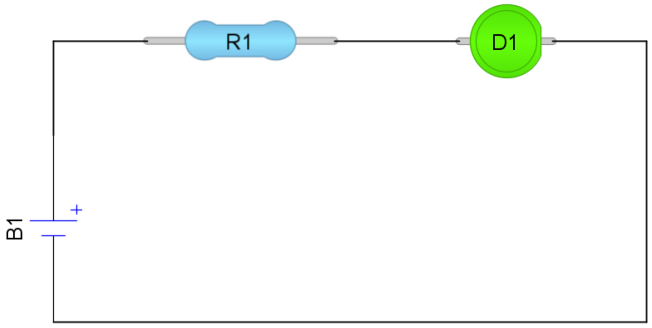
LEDs offer several advantages over traditional light sources such as incandescent bulbs. They are highly energy-efficient, converting a significant portion of electrical energy into light with minimal heat generation. LEDs also have a long lifespan, often exceeding 50,000 hours of operation. In addition to lighting, LEDs are used in optoelectronic applications such as remote controls, fiber-optic communication, and sensors. While LEDs offer many advantages, they require proper current regulation to avoid damage, as excessive current can lead to overheating and failure. This is typically managed using resistors or constant current drivers in LED circuits. We'll use resistors since they are cheap and easy to use, so let's explore how we can use the LED in the next lab.

Lab #5: Using the LED

In order to construct our circuit properly, we need to introduce a component called a resistor to limit the current flowing through our circuit. A resistor as we know is a component that can limit the flow of electrons though the LED. Connect the circuit as follows.



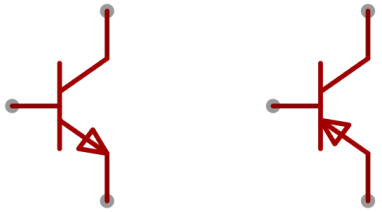
Now when you connect your circuit, you will observe that the LED stays lit. If we did not put the LED in the circuit, then the LED would burn out and become damaged. This is the way to create a proper load for our circuit. We can represent our breadboard circuit like this as well:



We will of course use the CH32V drive the LED though the resistor, this is just to make sure that everything works as expected.

Electronic Concept: Transistors

A transistor is one of the most revolutionary components in electronics, known for its ability to amplify signals and switch electrical currents. Its invention marked the beginning of the modern electronics era, making possible everything from compact radios to advanced computers. A transistor is a semiconductor device with a relatively simple structure but immense versatility, allowing it to function as the backbone of both analog and digital circuits.



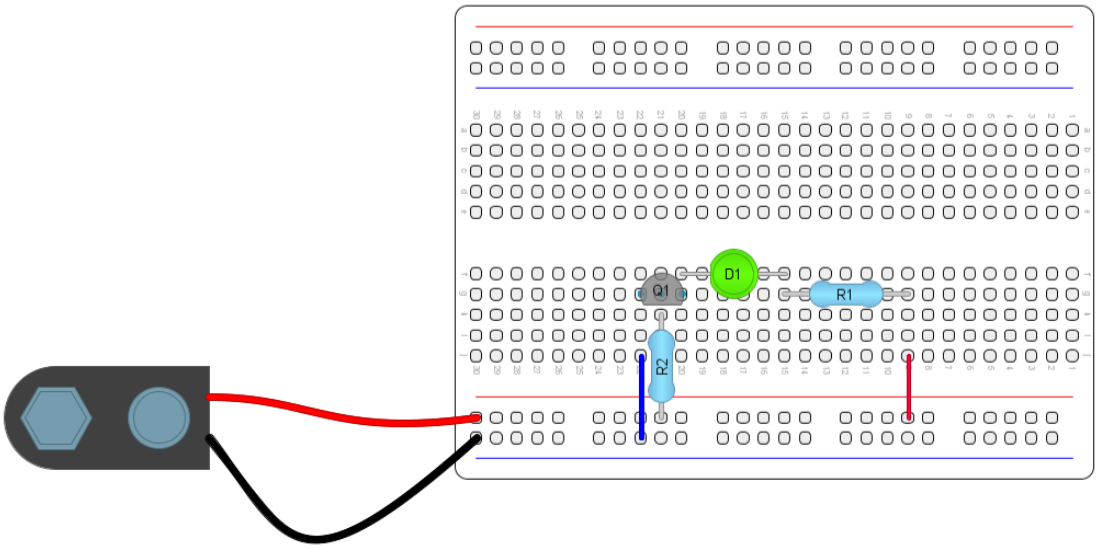
Transistor Schematic Symbols

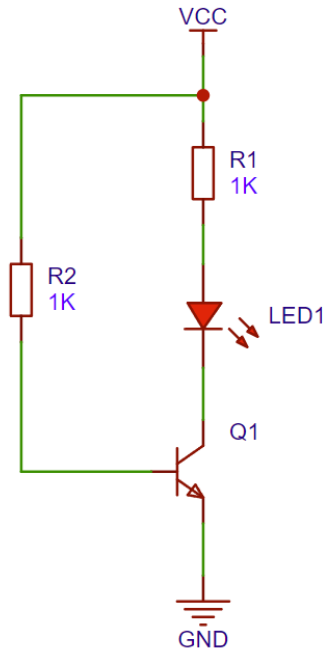
The structure of a transistor is based on semiconductor materials, such as silicon, arranged to form three distinct layers. These layers create two p-n junctions, forming three terminals, a base also known as the control terminal which is the straight line in the center of the two other pins, the emitter the terminal from which current flows out, you can identify it by the arrow on its pin and a collector that well collects the current. Transistors come in two main types which are Bipolar Junction Transistors (BJTs), comprised of either NPN or PNP semiconductor layers.

The operation of a transistor relies on the ability to control a large current flowing between the collector and emitter by applying a small current or voltage to the base (in BJTs) or gate (in FETs). This makes transistors ideal for amplification and switching applications. We will focus primarily on switching applications cause as we will see, in modern circuits op amps have largely replaced transistors for amplification tasks, except in specialized radio frequency (RF) and power amplification. In amplification, a transistor takes a small input signal and produces a larger output signal. For example, in audio systems, transistors amplify weak microphone signals to drive speakers. A small current at the base controls a larger current between the collector and emitter (in BJTs). This results in the amplification of the input signal, with the output being proportional to the input. In switching applications, a transistor operates in two states: on (conducting current) and off (blocking current). This switching ability is fundamental to digital electronics, where transistors serve as the building blocks of logic gates, memory cells, and microprocessors. In an NPN BJT, applying a small voltage to the base allows current to flow from the collector to the emitter, turning the transistor "on." When the base voltage is removed, the current stops, turning the transistor "off."

Lab #6: NPN Transistor Switch

We can use the 2N3904 NPN transistor as a switch, if you connect this circuit, the LED will be on, as soon as disconnect R2 at the transistor base the LED will turn off.



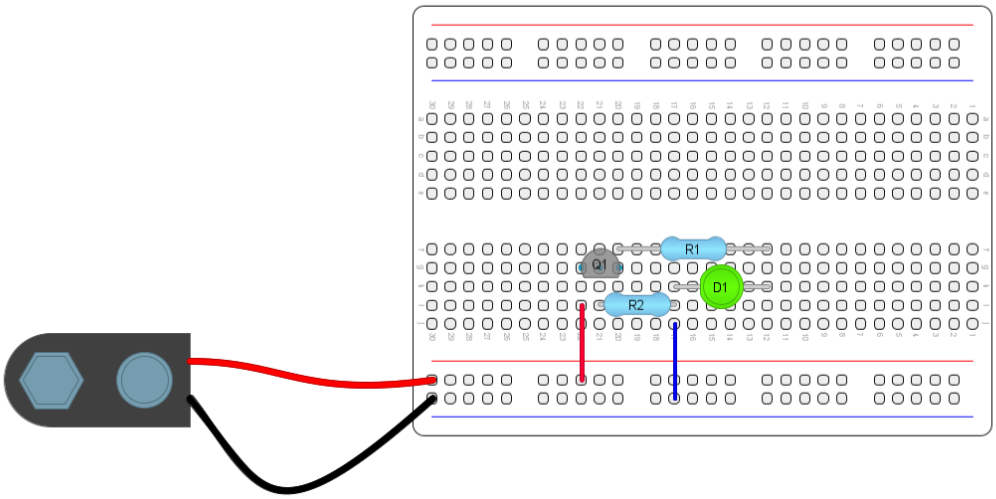


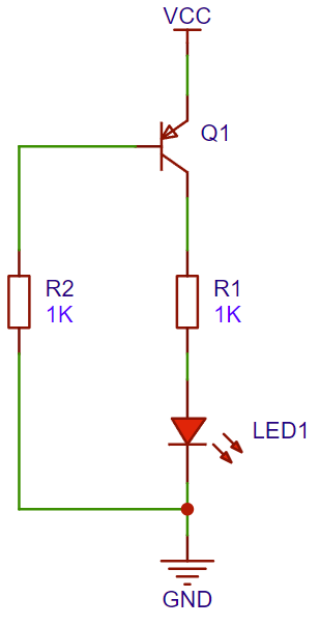
Connection Instructions:

- Connect VCC (positive rail) to one side of the 10kΩ resistor (R3).
- Connect the other side of R3 to the anode (long leg) of LED2.
- Connect the cathode (short leg) of LED2 to the collector (top pin) of the NPN transistor (Q2, 2N3904).
- Connect the emitter (bottom pin) of Q2 directly to GND.

Lab #7: PNP Transistor Switch

We can also use our 2N3906 PNP transistor as a switch, if you connect this circuit, the LED will be on, as soon as disconnect R2 at the transistor base the LED will turn off.



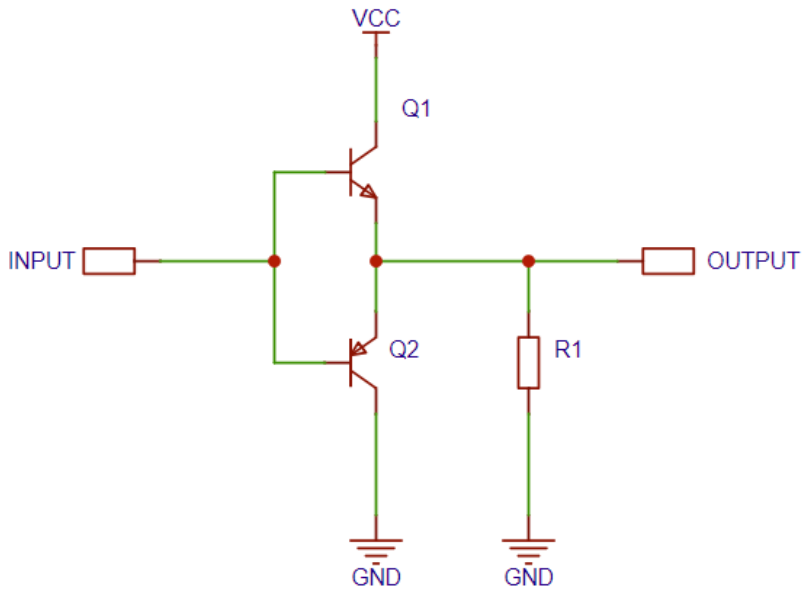


Connection Instructions:

- Connect VCC (positive rail) to the emitter (top pin) of the PNP transistor (Q1, 2N3906).
- Connect the collector (bottom pin) of Q1 to the anode (long leg) of the LED (LED1).
- Connect the cathode (short leg) of LED1 to GND (ground rail).
- Connect a 10kΩ resistor (R2) between VCC and the base (middle pin) of Q1.
- Connect a 1kΩ resistor (R4) between the base of Q1 and GND.

Electronic Concept: Push-Pull

You may be wondering what all this transistor stuff has to do with RISC-V microcontrollers. Bear with me, we're getting to that. This course is about understanding the underlying electronics remember? You may or may not have used transistors before so I'm trying to cater to all readers. Now that you understand how NPN and PNP transistors work intuitively, look at this circuit:

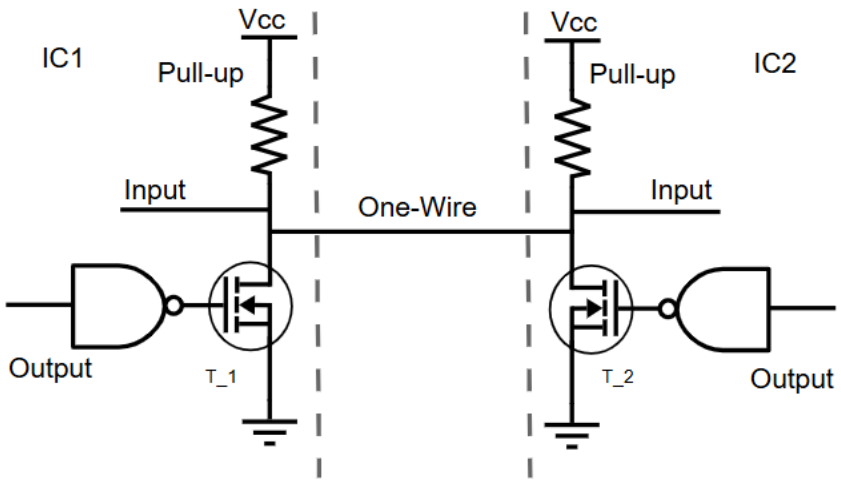


An electronic circuit in which two transistors (or vacuum tubes) are used, one as a source of current and one as a sink, to amplify a signal is called a "push-pull" circuit. One device "pushes" current out into the load, while the other "pulls" current from it when necessary.

Push-pull functionality allows the pin to drive both high and low logic levels. Traditionally we get this done by using two transistors, one connected to the supply voltage and the other to ground. When the pin is set high the upper transistor turns on pulling the pin up to Vcc and when its set low, the lower transistor is activated pulling the pin down to ground. This gives strong drive capability and we typically see them in the power stage of power electronics and audio circuits. Things like DC-DC converters, H-Bridge Motor Drivers and class AB amplifiers all use this configuration since it allows for symmetrical current handling capability as well as the ability to handle output swing.

It's also used in microcontrollers like the CH32V003 internally. Let's look at some CH32V GPIO internals to understand this a bit better.

The CH32V GPIO can be configured for multiple mode input or output functions. Which can be configured for push-pull or open drain function. In software you will see this written as GPIO_Mode_Out_PP or GPIO_Mode_Out_OD. Now you know about push-pull, but what the heck is open drain? Take a look at this image.



In a typical open-drain configuration, as illustrated, both devices (IC1 and IC2) have transistors that connect their outputs to ground when activated (these transistors here are a special type called MOSFETs, we'll recap them a bit later on in case you never worked with them before, but for our purposes they work just like transistors). When both outputs are high (i.e., transistors T_1 and T_2 are off), no current flows through the transistors, and the voltage on the shared one-wire bus is pulled up to Vcc by the external pull-up resistors. As a result, the input pins on both devices read a logical high. However, if either IC pulls its output low (activating its transistor), the line is pulled down to ground. This causes both inputs to register a logical low. Importantly, the use of pull-up resistors limits current flow, ensuring that there is no direct short between Vcc and ground even when the line is pulled low. This arrangement allows multiple devices to share a common communication line safely without conflict.

The GPIO also has pull-up and pull-down resistors that which we can enable or disable programmatically and can be used to add a lot of functionality to the pin.

CH32V GPIO Modes (pinMode())

In total we can put the pins on a port in the following configurations:

Floating input	An unconnected input pin that can pick up noise and randomly switch between high and low states.
Pull-up input	An input connected to Vcc through a resistor to ensure it reads high when no active signal is present.
Pull-down input	An input connected to ground through a resistor to ensure it reads low when no active signal is present.
Analog input	An input pin used to measure a continuous voltage level within a specified range using an ADC (Analog-to-Digital Converter).
Open-drain output	An output that can pull a line low but requires an external pull-up resistor to produce a high level.
Push-pull output	An output that can actively drive both high and low levels using two internal transistors.
Multiplexing the inputs and outputs of functions	Sharing one physical pin among multiple internal functions or signals, controlled by configuration registers.

When we multiplex functions, we call this giving the pin an "alternate function". Alternate functions can be both open drain we write as GPIO_Mode_AF_OD or push-pull GPIO_Mode_AF_PP.

Limits On Our GPIO

Take a look at this table excerpt from the datasheet.

Table 3-16 General-purpose I/O static characteristics

Symbol	Parameter	Condition	Min.	Typ.	Max.	Unit
V_{IH}	Standard I/O pin, input high level voltage		0.22*(V_{DD}-2.7)+1.55		V_{DD}+0.3	V
	FT I/O pin, input high level voltage		0.22*(V_{DD}-2.7)+1.55		5.5	V
V_{IL}	Standard I/O pin, input low-level voltage		-0.3		0.19*(V_{DD}-2.7)+0.65	V
	FT I/O pin, input low-level voltage		-0.3		0.19*(V_{DD}-2.7)+0.65	V
V_{hys}	Schmitt trigger voltage hysteresis		150			mV
I_{lkg}	Input leakage current	Standard I/O port			1	uA
		FT I/O port			3	
R_{PU}	Weak pull-up equivalent resistance		35	45	55	kΩ
R_{PD}	Weak pull-down equivalent resistance		35	45	55	kΩ
$C_{I/O}$	I/O pin capacitance			5		pF

Output drive current characteristics

GPIO (General-Purpose Input/Output Port) can sink or output up to ±8mA current, and sink or output ±20mA current (not strictly to V_{OL}/V_{OH}). In user applications, the total driving current of all I/O pins cannot exceed the absolute maximum ratings given in Section 3.2:

We see V_IH which is the input high voltage, that is the voltage that will be recognized as a logic high. The V_OL is input low voltage what will be recognized as a logic low. We also see that the Schmitt trigger voltage hysteresis is 150 mV. This defines a noise margin: the input must go above a higher voltage to register as high and below a lower voltage to register as low, helping with noise immunity.

We see that the input pins leak a small current when not actively driven as well as weak pull-up and pull-down resistances to keep the pin at a defined logic level when not actively driven. The table is telling is to keep if on the safe side we can source or sink up to 8 mA of current per pin and the total current from all I/Os must stay within the limits of the device.

From that we can also see that if too many I/Os drive high simultaneously at heavy loads, then the voltage may drop below expected levels due to high power ground currents. All In all though once we respect the limits of the device, the device outputs will maintain good logic levels low for '0' and high for a '1' under the expected loading and from this datasheet we see that the 5V tolerant inputs allow interfacing with 5V logic even if the core voltage is lower, though generally I advise to make things easy as a beginner, just keep your core voltage the same as your logic levels.

GPIO Pin Speed

You should also be aware that you can change the speed of the GPIO pins on your device. We adjust the speed using the MODEy configuration register and from the datasheet we see that the GPIO pin speed can by 2 MHz, 10 MHz or 30 MHz maximum speed settings. Fortunately, in the library provided by WCH we can easily see the definitions for the pin speeds in the GPIOSpeed_TypeDef enum.

```
/* Output Maximum frequency selection */
typedef enum
{
    GPIO_Speed_10MHz = 1,
    GPIO_Speed_2MHz,
    GPIO_Speed_30MHz
} GPIOSpeed_TypeDef;
```

Now you may be wondering why you would not just use the highest available speed. The reason is that when we set the pin to a higher speed, it actually increases power draw. If we run the device at lower speed, then we will have reduced power consumption which is useful in battery powered applications. Also, when you being looking at signal integrity you will see that high speed transitions can cause ringing, overshoot and EMI (Electromagnetic Interference) especially with long PCB traces or If we have sub-optimal layout.

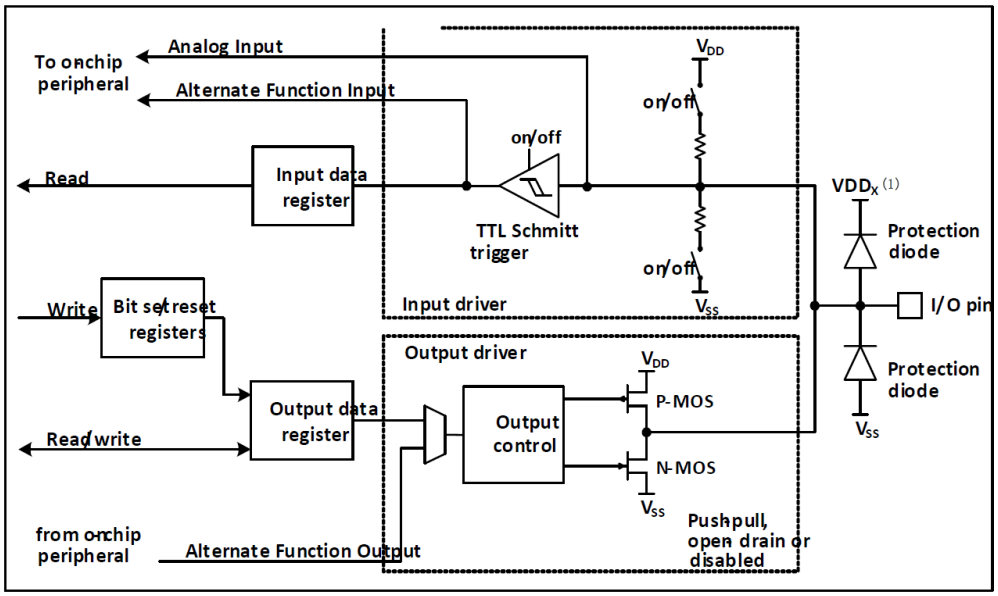
In general, you will set the pin speed to match the application. If you are doing basic I/O tasks like reading a pushbutton or turning on an LED, then you will want low speed. If however, you are doing something like pulse width modulation or using communication protocols you will need to set the pin speed high.

5-Volt Tolerant Pins

When using the pins remember that pins PC1, PC2, PC4, PC5 and PC6 are 5v tolerant pins. We can tell the 5-volt tolerant pins on CH32V devices because if we look at the datasheet, we see there is 'FT' in the pin type definitions section:

16	10	7	-	PC0	I/O	PC0	T2CH3	NSS_1/UTX_3/T2CH3_2 /T1CH3_1
1	11	8	5	PC1	I/O/FT	PC1	SDA/NSS	T1BKIN_1/T2CH4_1 T2CH1ETR(1)_2/URX_3 /T2CH1ETR(1)_3/T1BKIN_3
2	12	9	6	PC2	I/O/FT	PC2	SCL/URTS/T1BKIN	AETR_1/T2CH2_1 /T1ETR_3/URTS_1 /T1BKIN_2
3	13	10	-	PC3	I/O	PC3	T1CH3	T1CH1N_1/UCTS_1 /T1CH3_2/T1CH1N_3
4	14	11	7	PC4	I/O/A	PC4	T1CH4/MCO/A2	T1CH2N_1/T1CH4_2 /T1CH1_3
-	15	12	-	PC5	I/O/FT	PC5	SCK/T1ETR	T2CH1ETR(1)_1/SCL_2 /SCL_3/UCK_3/T1ETR_1 /T1CH3_3/SCK_1
5	16	13	-	PC6	I/O/FT	PC6	MOSI	T1CH1_1/UCTS_2/SDA_2 /SDA_3/UCTS_3/T1CH3N_3 /MOSI_1
6	17	14	-	PC7	I/O	PC7	MISO	T1CH2_1/URTS_2 /T2CH2_3/URTS_3 /T1CH2_3/MISO_1
7	18	15	8	PD1	I/O/A	PD1	SWIO/T1CH3N/AETR2	SCL_1/URX_1/T1CH3N_1 /T1CH3N_2
-	19	16	-	PD2	I/O/A	PD2	T1CH1/A3	T2CH3_1/T1CH2N_3 /T1CH1_2
-	20	17	-	PD3	I/O/A	PD3	A4/T2CH2/AETR/UCTS	T2CH2_2/T1CH4_1

Looking At Our Full GPIO



If we take a look at the right of the pin structure, we will see that we have the I/O pin which connects the microcontroller internals to the outside world. On this pin there are two protection diodes connected to VDD and VSS to protect against voltage spikes by clamping the pin voltage within limits.

We also have an input driver path on the top half of the device. When configured as an input the signal passes through a TTL Schmitt trigger which filters the noise and ensures clean logic level detection. The input data register captures the digital level and makes it available for the software to read. We also see the optional pull-up or pull-down resistors than can be enabled or disabled via software to prevent the pin from floating. From the block diagram we see that this path can also feed signals to internal analog peripherals like ADCs or alternate function inputs which will be things like or timers or modules for communication.

Complementary to this is the output driver path on the bottom half. When we configure it as output, the software writes data to the output register, which controls the output control block. The output stage uses a complimentary MOS pair, a P-MOSFET to drive the line high and an N-MOSFET to drive the line low. This forms our push-pull driver but can also be configures as a open drain or disabled in software.

I want you to pay attention to the Bit Set/Reset register when allows for efficient bit manipulation, letting the user set or clear specific bits without needing a read-modify-write cycle. We can also switch the pin to alternate function modes, connecting it to other on-chip peripherals such as UART, SPI, I2C, timers etc. instead of acting as a general-purpose pin.

Controlling The GPIO with C

At this point you know enough about GPIO pins to start using then in your programs. For output functions you will use the following C constructs. Now in C these depending on your library may be defined as function like macros, so lets take the GPIO_SetBits construct that will be used to control the pins. We can declare it like a function like macro like this:

```
#define GPIO_SetBits(GPIOx, PinMask) ((GPIOx)->ODR |= (PinMask))
```

In this case a preprocessor directive will get substituted at compile time as there is no type checking which makes this way or controlling the port registers very fast at the cost of safety.

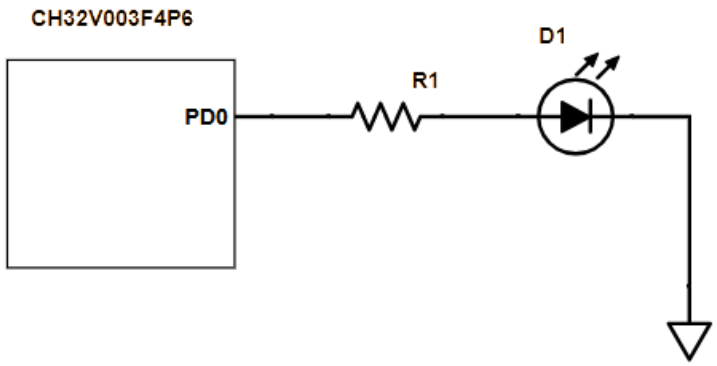
You can also have inline functions that are defined like:

```
static inline void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t PinMask) { GPIOx->ODR |= PinMask; }
```

In this case our definition will be type-safe, some compilers may also inline these in order to increase their speed. Regardless of how its implemented, you will typically use the following constructs to control your microcontroller port:

GPIO_SetBits	used for setting the GPIO High (1)	GPIO_SetBits(GPIOD, GPIO_Pin_0);
GPIO_ResetBits	used for setting the GPIO Low (0)	GPIO_ResetBits(GPIOD, GPIO_Pin_0);
GPIO_WriteBit	used when you want to SET (High) and REST (Low) the GPIO with the same function	GPIO_WriteBit(GPIOD, GPIO_Pin_0, SET); GPIO_WriteBit(GPIOD, GPIO_Pin_0, RESET); GPIO_WriteBit(GPIOC, GPIO_Pin_0, 1);
GPIO_Write	to set the complete GPIO port	
GPIO_PinLockConfig	Helps you lock the pin so that accidentally in other parts of the configuration can't be changed	
GPIO_ReadInputDataBit	used to get the status of a single pin (high or low)	u8 pin_status = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_3);
GPIO_ReadInputData	used to read the full port	

Lab #8: Blink (digitalWrite() and delay())



This lab is rather simple. On the hardware side, an LED is connected to pin PD0 of the CH32V003F4P6 through a 1 kΩ resistor (R1) to limit current, with the LED's cathode connected to ground. Driving PD0 high allows current to flow and turns the LED on, while driving it low turns the LED off.

If you are coming from an Arduino background, the software may initially seem daunting because the hardware setup is no longer hidden. Instead of using functions like pinMode() and digitalWrite(), the program explicitly configures the system and peripherals. The system initialization code sets up interrupt priorities, synchronizes the software with the microcontroller's 48 MHz internal clock, and initializes a delay mechanism. The GPIO configuration then enables the clock for port D and configures PD0 as a push-pull output, making the pin's electrical behavior clear and intentional.

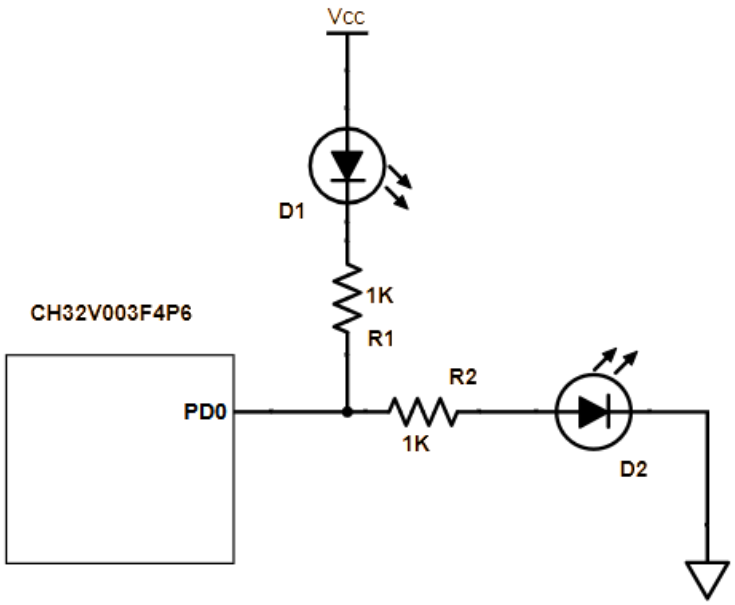
The LED blinking behavior is implemented by manually setting and clearing the PD0 output and inserting delays between each change.

The main() function initializes the system, configures the GPIO, and then enters an infinite loop that repeatedly toggles the LED. While this approach requires more setup than Arduino, it provides direct insight into how the microcontroller operates at the hardware level and highlights the level of control available when programming without abstraction layers.

```
// Includes
#include "debug.h"
// Initialization for the Main Program
void initMain()
{
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
   SystemCoreClockUpdate();
   Delay_Init();
}

// Initialize the GPIO
```

```
void initGPIO()
{
   // initialize structure used to hold GPIO config settings
   GPIO_InitTypeDef GPIO_InitStructure = {0};

   // Enable clock for GPIOD, clock needs to be enabled for Reset and Clock Control (RCC) unit
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
   // select pin 0 of PORTD to configure
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
   // Set the pin output to push-pull mode
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
   // seet outpit speed to 30 MHz
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
   // Apply the configuration to GPIOD using the settings provided in the init structure
   GPIO_Init(GPIOD, &GPIO_InitStructure);
}

// Function to blink the LED connected to PD0
void blinkLed()
{
   // set PIND0 low
   GPIO_WriteBit(GPIOD, GPIO_Pin_0, Bit_SET);
   Delay_Ms(250);
   // set PIND0 high
  GPIO_WriteBit(GPIOD, GPIO_Pin_0, Bit_RESET);
   Delay_Ms(250);
}

// Main Program
int main(void)
{
   // initialization
   initMain();
   initGPIO();

   while(1)
   {
    // main loop
    blinkLed();
   }
}
```

Lab #9: Dual LEDs on One Pin



The CH32V003F4P6 is a relatively low pin count device and because of that we can use techniques to optimize our I/O usage. We can drive two LEDs on a single pin provided you only want to switch on one LED at a time by exploiting the ability of the microcontroller to both sink and source current. In this project we extend our exploration of GPIO pins by using a single microcontroller pin to drive two LEDs in opposite directions. This design demonstrates the difference between a pin configured to sink current versus source current. The schematic places two LEDs, D1 and D2, with their respective current-limiting resistors R1 and R2, tied to the same GPIO pin. By toggling the output state of the pin between LOW and HIGH, we can alternately light one LED or the other.

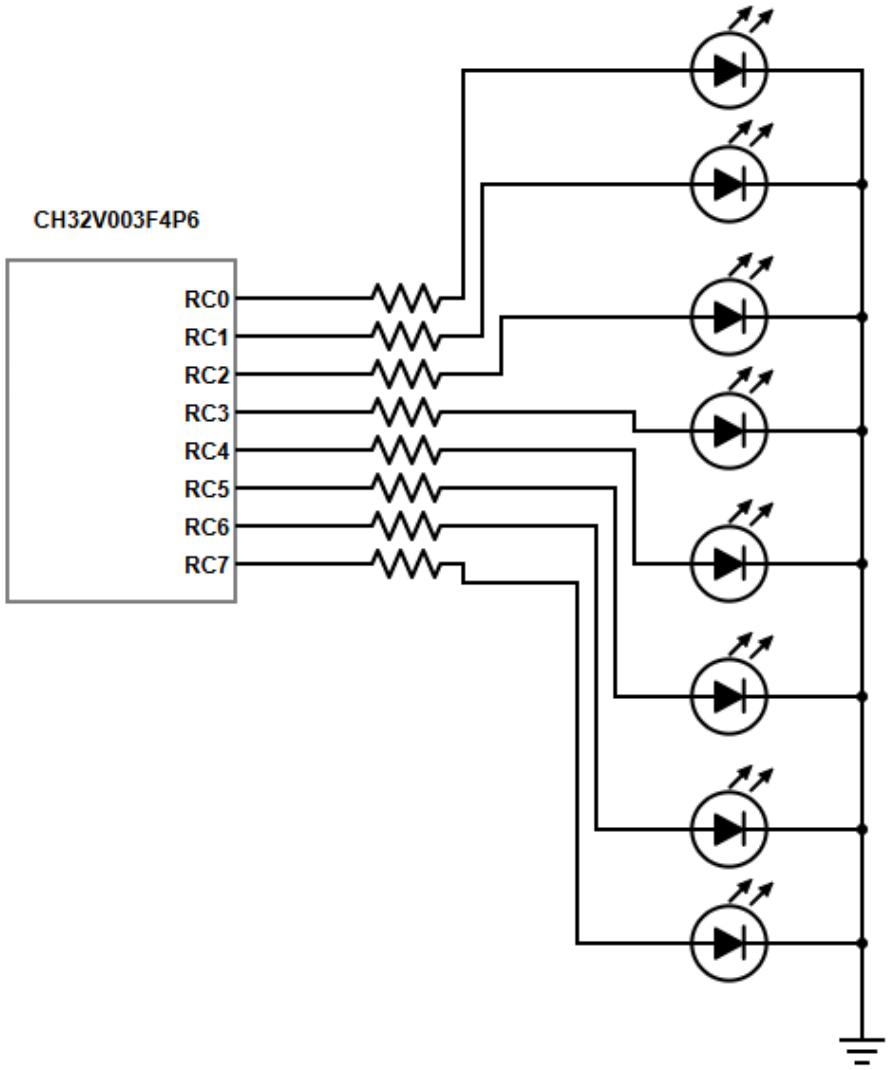
When the microcontroller output is driven LOW, the pin is effectively connected to ground potential. In this state, current flows from the positive supply (VCC), through diode D1, then through resistor R1, and finally into the microcontroller pin. Because the MCU is providing a ground reference, it is said to be sinking current. The current through D1 is limited by R1, causing D1 to illuminate. At the same time, diode D2 remains off because both its anode (connected through R2 to the GPIO pin) and its cathode (ground) are at nearly the same potential. With no voltage difference across it, D2 does not conduct, so it remains dark.

Lab #9 Listing

```
/*****************************************************************************
*Includes and defines
*****************************************************************************/
#include "debug.h"

typedef enum {
LED_A_ON, // Sink current: PD0 = LOW
LED_B_ON, // Source current: PD0 = HIGH
LED_OFF // Hi-Z: PD0 = input, both OFF
} LED_Mode;

void setLedMode(LED_Mode mode);
void initGPIO(void);

/*****************************************************************************
* Function: void initMain()
*
* Returns: Nothing
*
* Description: Contains initializations for main
*
* Usage: initMain()
*****************************************************************************/
void initMain()
{
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  SystemCoreClockUpdate();
  Delay_Init();
}

/*****************************************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
*****************************************************************************/
int main(void)
{
  // initialization
  initMain();
  initGPIO();

  while(1)
  {
    setLedMode(LED_A_ON);
    Delay_Ms(1000);

    setLedMode(LED_B_ON);
    Delay_Ms(1000);

    setLedMode(LED_OFF);
    Delay_Ms(1000);
  }
}

/*****************************************************************************
* Function: void initGPIO()
*
* Returns: Nothing
*
```

```
* Description: Contains initializations for GPIO pins
*
* Usage: initGPIO()
*******************************************************************************/
void initGPIO(void)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
}

/*******************************************************************************
* Function: void setLedMode(LED_Mode mode)
*
* Returns: Nothing
*
* Description:
*
* Usage: setLedMode(LED_x_ON);
*******************************************************************************/
void setLedMode(LED_Mode mode)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;

  if (mode == LED_OFF)
  {
    // High impedance mode: turn both LEDs off
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOD, &GPIO_InitStructure);
  }
  else
  {
    // Enable output mode
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    // Set output level
    GPIO_WriteBit(GPIOD, GPIO_Pin_0, (mode == LED_B_ON) ? Bit_SET : Bit_RESET);
  }
}
```

Lab #10: Bidirectional Knightrider



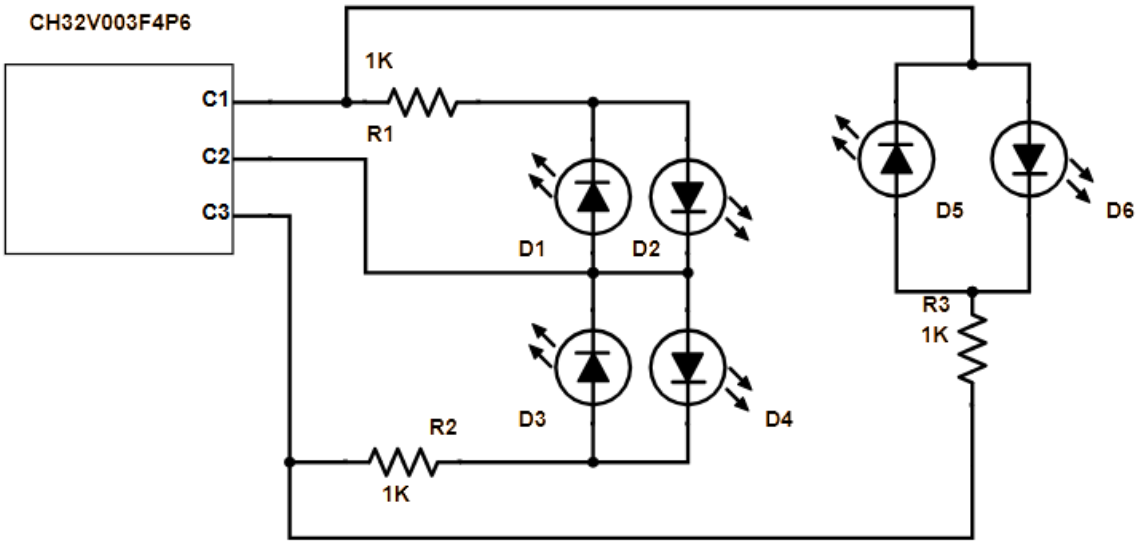
We can use the CH32V003 to create knight rider LEDs. This is a popular LED configuration where the lights sweep back and forth in a sweeping pattern and comes from a 1980s TV series called Knight Rider. It's sometimes called the Larson scanner and is a good way to demonstrate outputs and direction control on our microcontroller port. We can connect LEDs to all the pins of PORTC in order to demonstrate the effect.

```c
/*****************************************************************************
*Includes and defines
*****************************************************************************/
#include "debug.h"

// --------- TUNE ME ---------
#define LED_ACTIVE_HIGH 1 // 1 = drive pin HIGH turns LED ON; 0 = drive pin LOW turns LED ON
#define STEP_DELAY_MS 60 // delay between steps

// List the pins you're using on PORTC, in order left->right on your light bar
static const uint16_t PIN_LIST[] = {
  GPIO_Pin_0, GPIO_Pin_1, GPIO_Pin_2, GPIO_Pin_3,
  GPIO_Pin_4, GPIO_Pin_5, GPIO_Pin_6, GPIO_Pin_7
};

#define LED_COUNT (sizeof(PIN_LIST)/sizeof(PIN_LIST[0]))

static void initGPIO(void);
static inline void leds_all_off(void);
static inline void led_one_on(size_t i);
static void knight_rider_step(void);


/*****************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*****************************************************************************/
static void initMain(void)
{
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
   SystemCoreClockUpdate();
   Delay_Init();
}

/*****************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*****************************************************************************/
int main(void)
{
   initMain();
   initGPIO();

   while (1) {
```

```
    knight_rider_step();
  }
}


/*******************************************************************************
* Function: static void initGPIO(void)
* Returns: Nothing
* Description: Initializes pins as push-pull @ 30 MHz can also be used
* active low
* Usage: initGPIO()
*******************************************************************************/
static void initGPIO(void)
{
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

  GPIO_InitTypeDef GPIO_InitStructure = {0};
  GPIO_InitStructure.GPIO_Pin =
    GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3 |
    GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  // All OFF to start
  #if LED_ACTIVE_HIGH
    GPIO_ResetBits(GPIOC, GPIO_InitStructure.GPIO_Pin);
  #else
    GPIO_SetBits(GPIOC, GPIO_InitStructure.GPIO_Pin);
  #endif
}


/*******************************************************************************
* Function: static inline void leds_all_off(void)
* Returns: Nothing
* Description: Turns all LEDs off
* Usage: leds_all_off()
*******************************************************************************/
static inline void leds_all_off(void)
{
  #if LED_ACTIVE_HIGH
   GPIO_ResetBits(GPIOC, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3 |
   GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7);
  #else
   GPIO_SetBits(GPIOC, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3 |
   GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7);
```

```
   #endif
}


/*******************************************************************************
* Function: static inline void led_one_on(size_t i)
* Returns: Nothing
* Description: Lights exactly one LED at index i
* Usage: led_one_on((size_t)i)
*******************************************************************************/
static inline void led_one_on(size_t i)
{
  uint16_t pin = PIN_LIST[i];
  leds_all_off();
  #if LED_ACTIVE_HIGH
    GPIO_SetBits(GPIOC, pin);
  #else
    GPIO_ResetBits(GPIOC, pin);
  #endif
}

/*******************************************************************************
* Function: static void knight_rider_step(void)
* Returns: Nothing
* Description: Knight Rider sweep: 0->N-1 then N-2->1
* Usage: knight_rider_step()
*******************************************************************************/
static void knight_rider_step(void)
{
  // forward
  for (size_t i = 0; i < LED_COUNT; ++i) {
    led_one_on(i);
    Delay_Ms(STEP_DELAY_MS);
  }
  // backward (skip the ends to avoid a double hold)
  for (int i = (int)LED_COUNT - 2; i >= 1; --i) {
   led_one_on((size_t)i);
   Delay_Ms(STEP_DELAY_MS);
  }
}
```

Lab #11: Complementary LED Drive



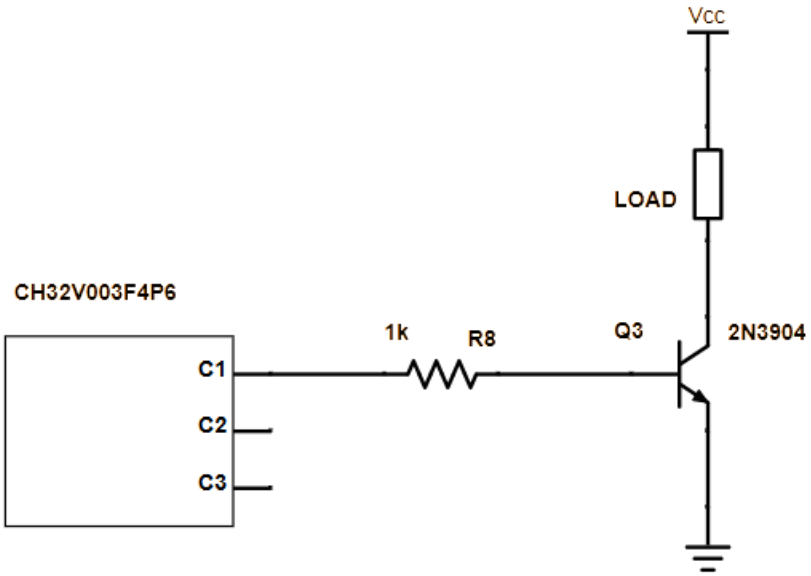
Building on the idea of driving dual LEDs on a single pin, there is a circuit known as the complementary LED drive where you can drive a large number of LEDs using the tri-state logic ability of our microcontroller pin. This type of multiplexing is also called “Charlieplexing”. Using this method, we can drive up to Six LEDs using just three I/O lines.

```
/*****************************************************************************
*Includes and defines
*****************************************************************************/
#include "debug.h"

#define LED_PIN_1 GPIO_Pin_1
#define LED_PIN_2 GPIO_Pin_2
#define LED_PIN_3 GPIO_Pin_3
#define LED_PORT GPIOC

typedef enum {
 LED_1 = 0,
 LED_2,
 LED_3,
 LED_4,
 LED_5,
 LED_6,
 LED_COUNT
} LEDIndex;

typedef struct {
 uint16_t source;
 uint16_t sink;
  uint16_t hiz;
} LEDControlMap;

LEDControlMap ledMap[LED_COUNT] = {
 {LED_PIN_1, LED_PIN_2, LED_PIN_3}, // LED 1
 {LED_PIN_2, LED_PIN_1, LED_PIN_3}, // LED 2
 {LED_PIN_1, LED_PIN_3, LED_PIN_2}, // LED 3
 {LED_PIN_3, LED_PIN_1, LED_PIN_2}, // LED 4
 {LED_PIN_2, LED_PIN_3, LED_PIN_1}, // LED 5
 {LED_PIN_3, LED_PIN_2, LED_PIN_1}, // LED 6
};


void enableLED(LEDIndex led);
void set_pin(GPIO_TypeDef* port, uint16_t pin, GPIOMode_TypeDef mode, BitAction val);


/*****************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*****************************************************************************/
void initMain()
```

```
{
 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
 SystemCoreClockUpdate();
 Delay_Init();

 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
}


/*********************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*********************************************************************************/
int main(void)
{
  // initialization
  initMain();

  while(1)
  {
   for (LEDIndex i = LED_1; i < LED_COUNT; i++)
   {
     enableLED(i);
     Delay_Ms(1000);
   }
  }
}

/*********************************************************************************
* Function: void set_pin(GPIO_TypeDef* port, uint16_t pin,
* GPIOMode_TypeDef mode, BitAction val)
* Returns: Nothing
* Description: Contains initializations for main
* Usage: set_pin(LED_PORT, config.source, GPIO_Mode_Out_PP, Bit_SET);
* set_pin(LED_PORT, config.sink, GPIO_Mode_Out_PP, Bit_RESET);
* set_pin(LED_PORT, config.hiz, GPIO_Mode_IN_FLOATING, 0);
*********************************************************************************/
void set_pin(GPIO_TypeDef* port, uint16_t pin, GPIOMode_TypeDef mode, BitAction val)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};
  GPIO_InitStructure.GPIO_Pin = pin;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_InitStructure.GPIO_Mode = mode;
  GPIO_Init(port, &GPIO_InitStructure);

  if (mode == GPIO_Mode_Out_PP)
```

```
  {
    GPIO_WriteBit(port, pin, val);
  }
}


/*******************************************************************************
* Function: void enableLED(LEDIndex led)
* Returns: Nothing
* Description: Contains initializations for main
* Usage: enableLED(i)
*******************************************************************************/
void enableLED(LEDIndex led)
{
  LEDControlMap config = ledMap[led];

   set_pin(LED_PORT, config.source, GPIO_Mode_Out_PP, Bit_SET); // Source HIGH
   set_pin(LED_PORT, config.sink, GPIO_Mode_Out_PP, Bit_RESET); // Sink LOW
   set_pin(LED_PORT, config.hiz, GPIO_Mode_IN_FLOATING, 0); // Hi-Z
}
```

Lab #12: Low Side Transistor Switching



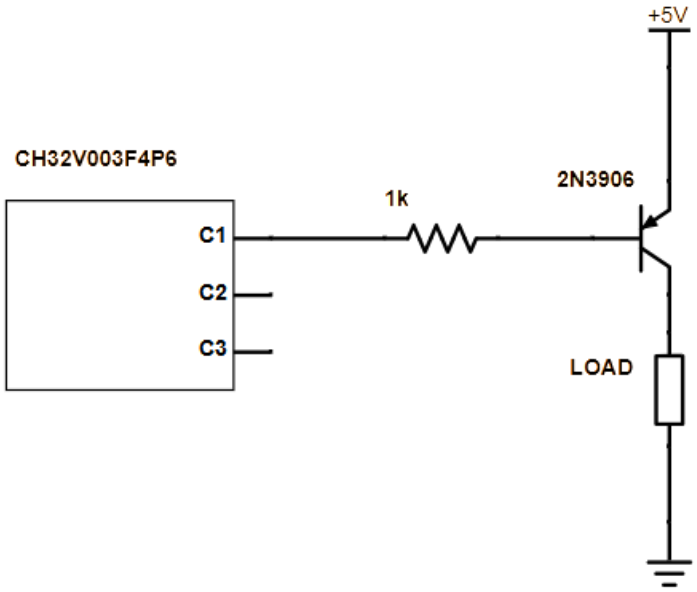
The pin on your CH32V003 can only handle up to 8mA max, but sometimes you may need to drive larger loads into the tens of milliamps. In such cases you need a low side signal transistor to handle this switching for you. Though persons may have their preferences, the most versatile device I think you can use is the 2N3904. We can use it to drive loads.

The reason for selecting this transistor is crucial, it's easy to reach the point of saturation. In order to get your transistor saturated, it must reach a state where even though the base current into the transistor is increasing, the collector current is not increasing anymore. When saturated, the transistor acts like a low-resistance switch, minimizing power loss and heat. When we are driving our devices with microcontrollers, we want our transistor to saturate within reason since the microcontroller pin can provide limited drive current. As such it's a good rule of thumb to design for a collector current 10 times greater than the base current.

So, for our example, assuming we are driving our circuit with a 3.3v power supply, the current at the base will be: IB = 3.3V – 0.7V / 1k = 2.6V / 1000 = 2.6 mA, then our max collector current is 10 x IB = 10 x 2.6 mA or 26 mA.

This means that the transistor will saturate up to about 25-30 mA of current. If the load requires more than 30 mA, then the 2N3904 won't fully saturate causing power loss and voltage drop across the transistor, making it heat up slightly and operate in the linear region, not ideal for switching. For that we'll need to look at devices that have stronger drive capability which we will look at later, but for now let's look at some more transistor switching.

Lab #13: High Side Transistor Switching

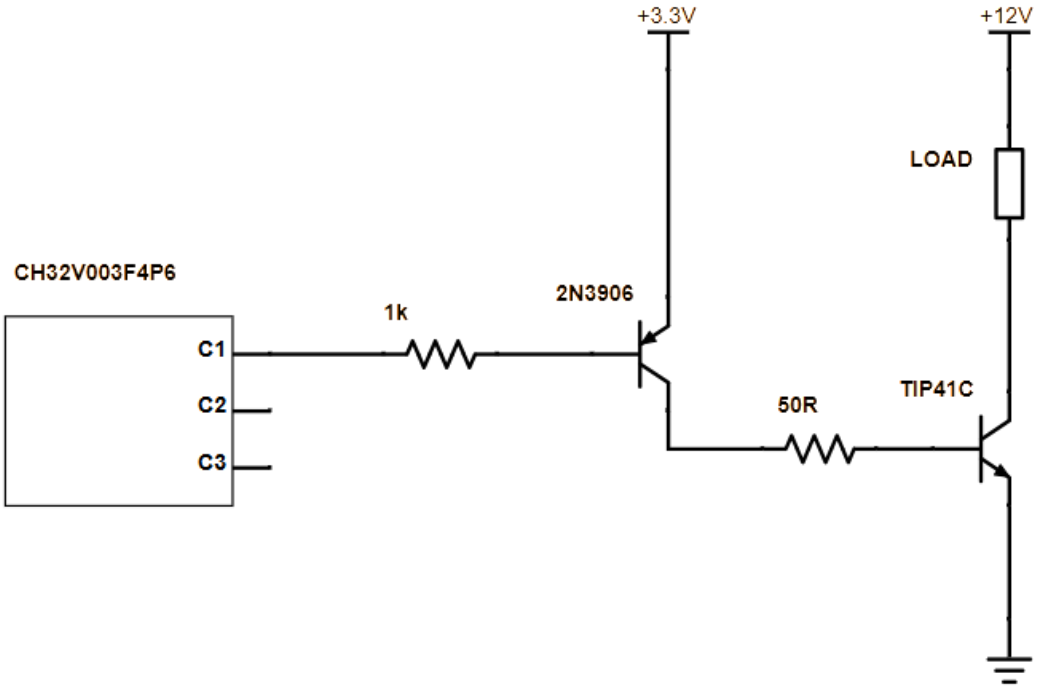


You can also use the microcontroller to switch loads high side using a 2N2906 transistor. It's very similar to the low side driver. The low side switching is very effective because it's simple, efficient and easy to drive from a microcontroller pin directly because the emitter is at ground potential.

However, in many real systems, the load must remain referenced to ground, and switching occurs on the positive supply rail instead. This is known as high-side switching, where the transistor is placed between Vcc and the load rather than between the load and ground. It's a bit more inefficient, so you may be asking yourself when would we ever use this?

Well, we typically use this when the ground connection of the load must remain permanently connected and cannot be interrupted. In many real systems ground is treated as a fixed reference shared by multiple circuits, sensors or safety-critical components. Disconnecting ground as is done in low side switching can cause unpredictable behavior, measurement errors, noise injection or even system faults. If you ever work in automotive systems where you typically want the chassis ground to always be present for loads like lamps, motors and solenoids, you would use high-side switching. In mixed-signal systems like the ones we are focusing on building, analog sensors can sometimes have signal corruption and noise when we switch the power line, so in some circuits we can keep the ground solid and preserve signal integrity and measurement accuracy.

Lab #14: Power Transistor Switching

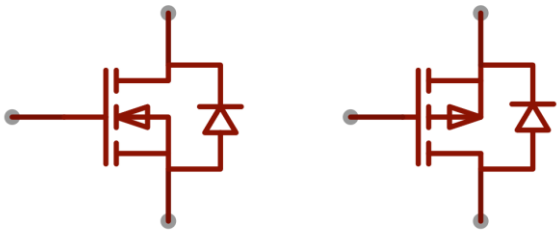


Power transistor switching is used when the load current exceeds what small-signal transistors and microcontroller pins can safely handle. Our earlier labs could switch tens of milliamps, however for real systems we typically need to drive motors, relays, solenoids, heaters or high-power LEDs. Here is a circuit you can use to switch a few hundred milliamps with comfortably. This setup in particular can switch about 0.5 A safely without any issues, of course you could replace it with something like a 2N2222A in those ranges, but it you increase the supply voltage of the 2N3906 to 5v this circuit will be able to switch up to 800 mA without much hassle and with a bit of tweaking up to about 1 A.

We can use this circuit in real world systems where we transition from logic-level control to real-world actuation. The proper name for this use case is voltage domain separation. Power transistor stages allow low-voltage controllers, such as 3.3 V microcontrollers, to control loads operating at higher voltages like 5 V, 9 V, or 12 V. This separation protects sensitive logic circuitry while allowing the system to interface with industrial or automotive power rails.

Electronic Concepts: The MOSFET

A MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor) is a type of transistor used extensively in electronics for switching and amplification. Its ability to handle high speeds, low power, and large currents makes it indispensable in a wide range of applications, from microprocessors and power supplies to motor control and audio amplifiers. The MOSFET's versatility, efficiency, and reliability have made it a cornerstone of modern electronics.



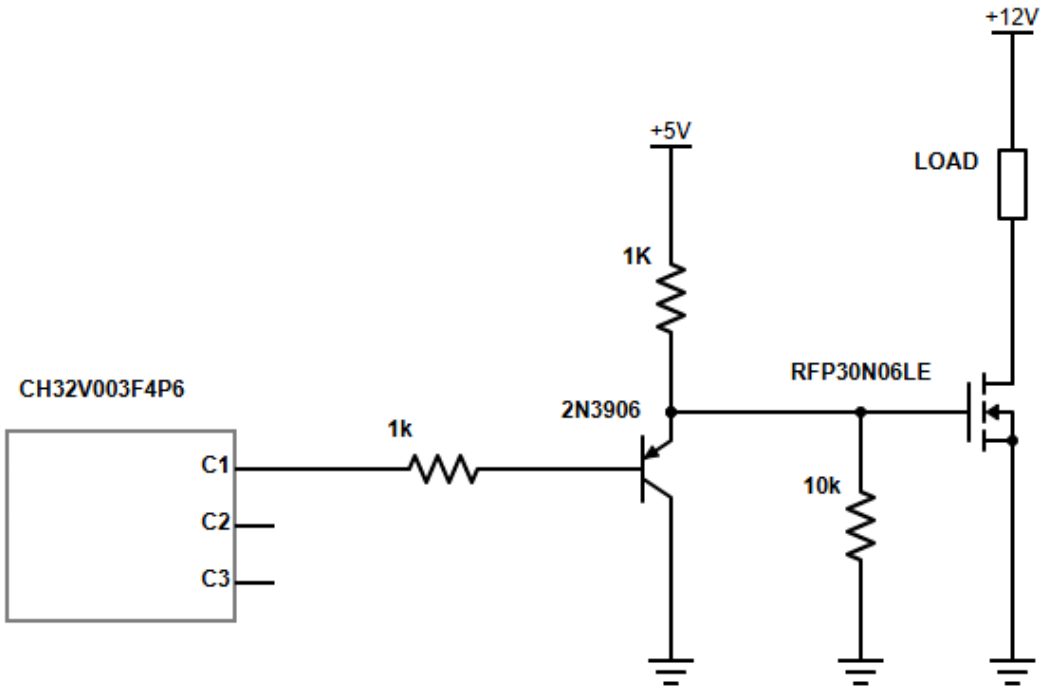
MOSFET Schematic Symbols

The operation of a MOSFET is based on controlling the flow of current between its drain and source terminals using a voltage applied to the gate terminal. The gate is separated from the channel (the region between the drain and source) by a thin insulating layer of silicon dioxide, allowing the gate to control the channel's conductivity without requiring a direct current. This high input impedance makes MOSFETs very energy-efficient, as they consume minimal power when switching.

MOSFETs are broadly categorized into two types: Enhancement-mode and Depletion-mode. When the MOSFET is in enhancement mode the MOSFET is normally off when the gate-to-source voltage is zero, which means that once we apply a suitable gate voltage, the channel conductivity will enhance, will will allow current to flow. Depletion mode devices however are normally on and applying a gate voltage of an opposite polarity will deplete the channel of carriers, which will reduce or stop current flow.

We can categorize MOSFETs into two categories N-Channel and P-channel devices. N-channel MOSFETs are more common due to their lower resistance and faster switching characteristics compared to P-channel devices. MOSFETs also come in two primary configurations: power MOSFETs for high-power applications and small-signal MOSFETs for low-power and high-frequency use.

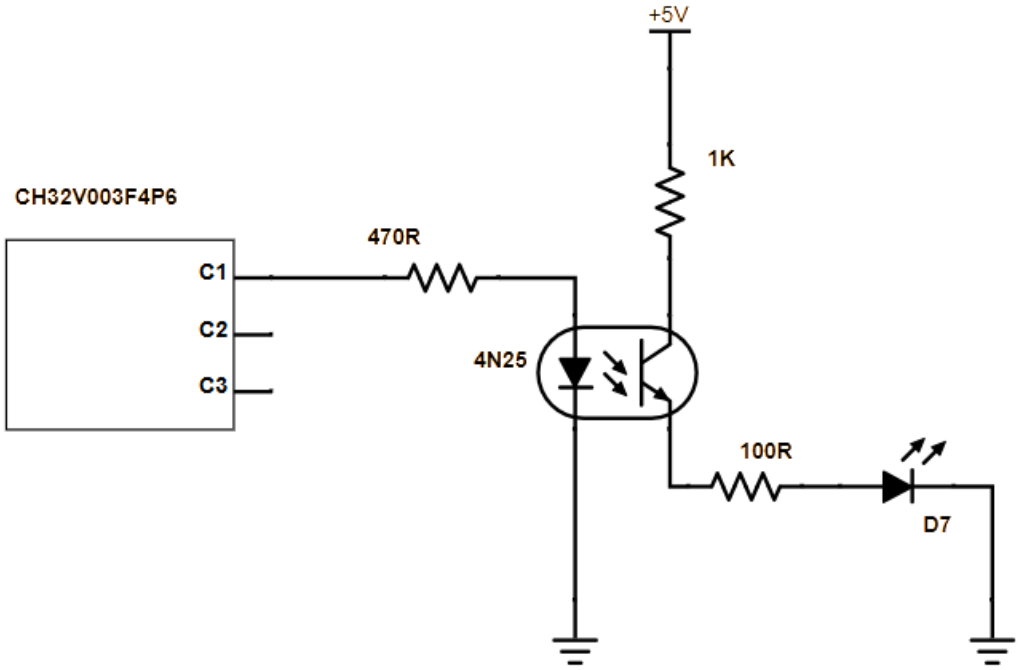
Lab #15: MOSFET Switching



If you need to control currents up to the 10-15A range, then you will have a hard time finding a transistor to handle that. In such cases you will need to use a MOSFET. The problem with MOSFETs is that they require driving in order to work properly. You can circumvent this problem by using a logic level MOSFET like the RFP30N06LE, or the IRL2910 or even the IRLZ44N and using a clever drive circuit.

By using a logic level MOSFET we can help with most of the driver issues by using a 2N3906 transistor to drive the MOSFET. This is an excellent PNP switch that allows use to use a 3.3v volt Microcontroller like the CH32V003 to control a 12V high power load. The 2N2906 or other PNP will act as a logic shiftier, shifting our 3.3v up to 5v. We then use it to control a 12v load. We limit the base current into the 2N3906 and add a pulldown on the MOSFET gate to turn it cleanly off. The PNP transistor is in emitter follower configuration. What this means is that the MOSFET will always be on unless we drive the pin low. For safe measure you can also add a pull-up to 5V at the base of the transistor. It is uncommon to see a PNP transistor used in this way, however by using the PNP as a controlled pull-up we achieve a clever and efficient way to use the MOSFET to control loads.

Lab #16: Isolated Switching

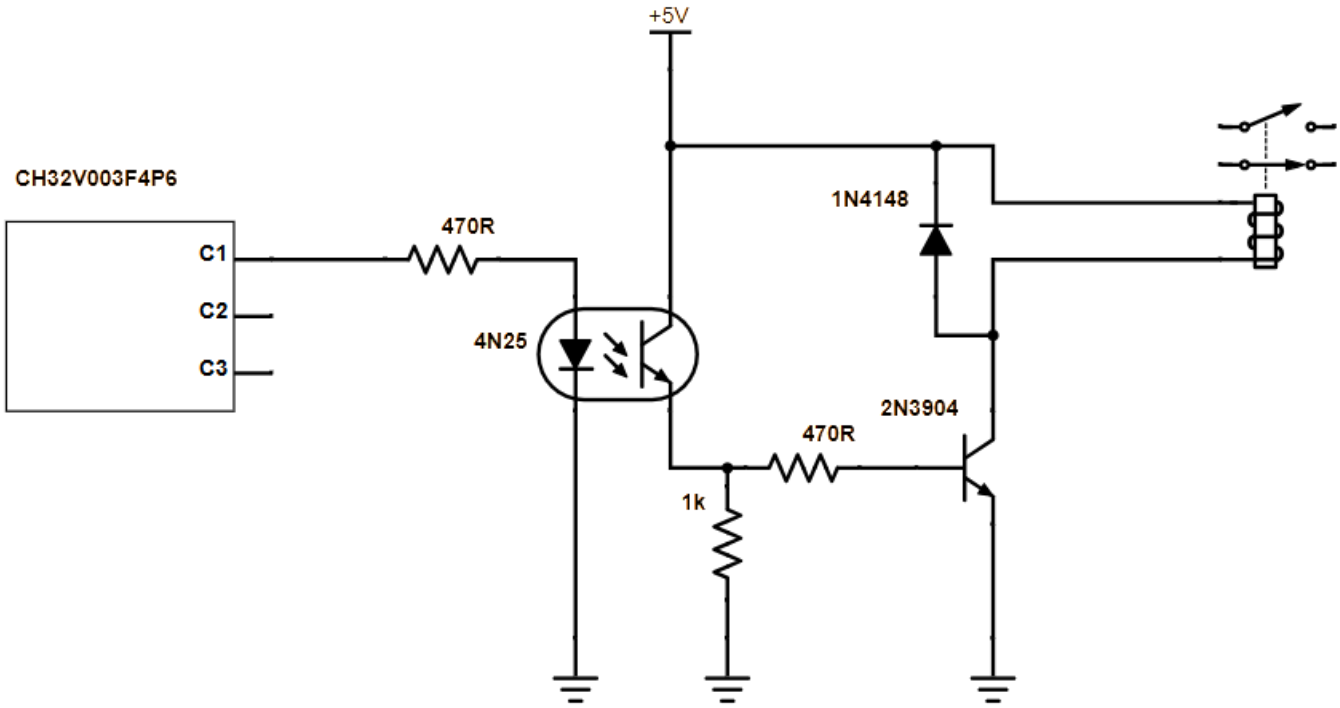


Isolated switching is used when a microcontroller must control a circuit that operates at a different electronical potential or in a noisy or hazardous environment, where a direct electrical connection will be unsafe or unreliable. In this approach the control signal is transferred using light rather than a direct electrical path, ensuring that no current, voltage spikes or disturbance can flow back into the microcontroller. This separation is known as galvanic isolation and is one of the most important protection techniques. In this type of control, the controlled circuit and the control logic do not share the same ground reference, or when sharing ground would introduce measurement errors or safety risks. We can perform isolated switching using an optocoupler. An optocoupler is used to galvanically isolate our microcontroller from any other current or voltage in its surroundings.

The choice of resistor from the CH32003F4P6 pin C1 into the optocoupler was not by guess. It's actually the optimal resistor value. Generally, the stronger the LED drive in an optocoupler it means faster switching and things like that. However, the current limitation of the CH32V003 places us at 6 mA, yes, we can draw more but there are other tradeoffs. That means that we must respect these limits. The typical forward voltage for the 4N25 is around 1.2 volts. We usually want a forward voltage of around 5-10 mA. The closest we can reach is 4.5 mA with a 470R resistor which will give us a safe and reliable ~4.5 mA current.

A 330R resistor will give us 6.4 mA which is slightly closer to the 8 mA limit on the pins, and anything lower will not make much sense since we will have a lower current without any benefit. We can switch at maybe 10-20 kHz max with this setup which is more than enough for many applications.

Lab #17: Switching Inductive Loads

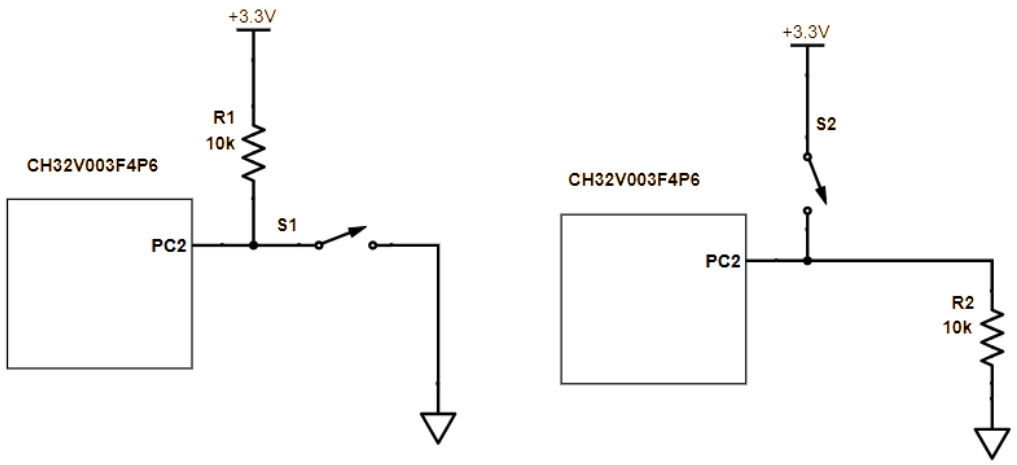


Once you begin to drive LEDs with your microcontroller it is expected that you will want to drive other things one of which is usually a DC motor or a relay. Motors and relays are what we call an inductive load. Inductive loads are dangerous because as we know inductors store energy like a capacitor only instead of an electric field, it does so using a magnetic field. When we switch off our circuit, this energy from the inductor needs to go somewhere, as the inductive filed collapses and creates a large spike that has very high voltage. We will look at driving motors later, but for now we will look at driving a relay which is an inductive load with similar properties to a motor.

To combat this problem, we typically add a clamping diode, also called a snubber diode across the inductive coil in order to limit the voltage spike that results from the collapsing magnetic field. Since the diode is helping with the conduction, a slower diode means it will take a longer time for magnetic field decay. The optoisolator is used to keep the MCU power supply stable, as sometimes driving an inductive load can drop the MCU supply voltage and protects the port. The 1k pulldown on the base of the transistor is needed because tiny leakages in the optoisolator can sometimes partially turn off the transistor. The relay I used were 5v relays with a coil resistance measure at 69 ohms. So the current draw is expect to be around 5/69 = ~ 72.5 mA coil current, and will consume around 5 x 0.0725 = ~ 0.36 W of power. If you're using a larger relay upgrade the transistor to a 2N2222A and upgrade the diode to a 1N4148 one.

If in doubt measure the coil resistance and perform the necessary calculations.

GPIO Inputs



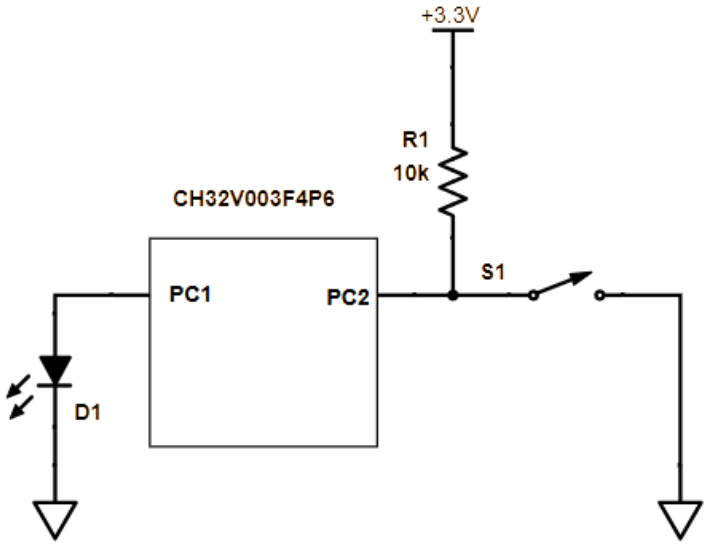
In the last section we looked at how we can perform outputs with the device. Up to 17 pins on the device can be used as I/O and 18 if we configure the reset pin to be used as a GPIO pin as well. I/O means the pins can perform not only output but also input. The first type of input we will look at is called polling input. Polling is a synchronous approach to handling input where the CPU actively checks for the state of our input in a loop. Typically, we do polling something like this:

```
while (1) {
 if (button_pressed()) {
  turn_on_LED();
 } else {
  turn_off_LED();
 }
}
```

In this code construct the CPU constantly checks whether the button is pressed. Now this is from the software side, however from the hardware side it is important to know that there are two main ways we can connect a pushbutton to a microcontroller pin using with a pull-up or pull-down configuration.

In pull-up configuration we use a resistor to connect the pin to our supply voltage which is shown on the left. In pull-down configuration the switch is connected between the power supply voltage and our microcontroller device pin as shown on the right.

Lab #18: Pushbutton (digitalRead())



We can set up a simple digital input circuit using a pull-up resistor configuration, as shown in the schematic. When the switch is open, the pull-up resistor forces the input pin to a logic HIGH, and when the switch is pressed, the input is connected to ground and reads as a logic LOW. This ensures the input never floats and always has a well-defined logic level.

Although the microcontroller provides internal pull-up resistors, these are typically weak and can only source a small amount of current. As a result, the signal may rise slowly and be more affected by noise, especially when using long wires or operating in electrically noisy environments.

Using an external pull-up resistor allows more current to flow into the input pin, producing faster rise times and cleaner signal transitions. This improves noise immunity and overall reliability, which is why external pull-ups are often preferred in practical and professional designs.

```c
/*********************************************************************
*Includes and defines
*********************************************************************/
#include "debug.h"
void initGPIO(void);


/*********************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*********************************************************************/
void initMain(void)
{
 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
 SystemCoreClockUpdate();
 Delay_Init();
}

/*********************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*********************************************************************/
int main(void)
{
  uint8_t buttonState = 0;
  initMain();
  initGPIO();

  while(1)
  {
   // Read button on PC2
   buttonState = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_2);

   // Active-low button logic (0 when pressed)
  if(buttonState == 0)
  {
   // Turn LED ON (PC1 = LOW if active-low LED)
   GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_RESET);
  }
  else
  {
    // Turn LED OFF
   GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_SET);
  }
```

```
    Delay_Ms(50); // Debounce delay
  }
}

/*******************************************************************************
* Function: void initGPIO()
* Returns: Nothing
* Description: Contains initializations for GPIO pins PC1 is setup as LED output
* and PC2 is setup as button input
* Usage: initGPIO()
*******************************************************************************/
void initGPIO(void)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};

  // Enable GPIOC Clock
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

  // --- Configure PC1 as Output Push-Pull ---
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  // --- Configure PC2 as Input with Pull-Up ---
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; // Input pull-up
  GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

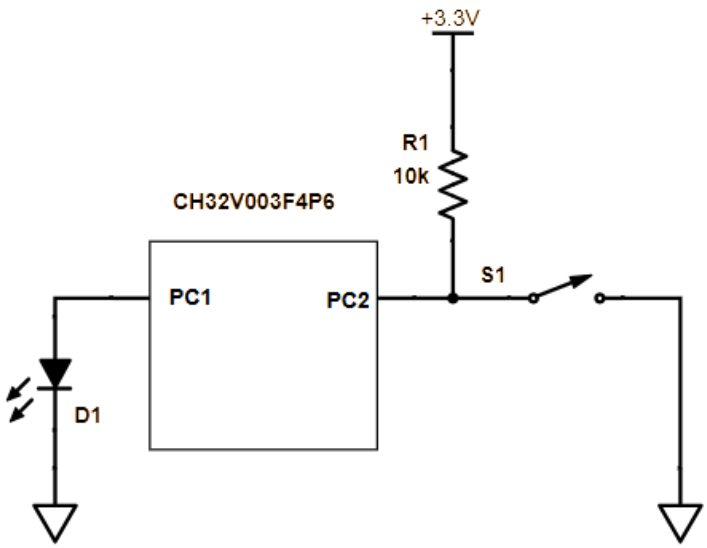
Button Debouncing

Switches are mechanical devices with moving parts and what that means is that there will be some problems inherently associated with that especially since the microcontroller devices operates at a much greater speed than switching devices you will interface with. When you press a pushbutton, the microcontroller will register many presses of the switch because there may be quite a number of micro openings and closing of the switch taking place in that short period of time. This is like when you drop a rubber ball, because of the physical properties of the material the ball is made from, the ball does not come to a rest as soon as it hits the ground but instead it bounces less and less until it finally settles.

This 'bouncing' of the switch before settling presents a risk that the microcontroller will read the switch as being open when the circuit designer expects it to be closed and vice versa. To counteract this problem you can implement switch

```
uint8_t debounceButton(GPIO_TypeDef* port, uint16_t pin)
{
  if (GPIO_ReadInputDataBit(port, pin) == 0)  // Button pressed (active-low)
  {
    Delay_Ms(20);  // Wait 20ms
    if (GPIO_ReadInputDataBit(port, pin) == 0)  // Still pressed?
    {
      while (GPIO_ReadInputDataBit(port, pin) == 0);  // Wait until released
      Delay_Ms(20);  // Confirm release
      return 1;
    }
  }
  return 0;
}
```

Lab #19: On/Off Pushbutton



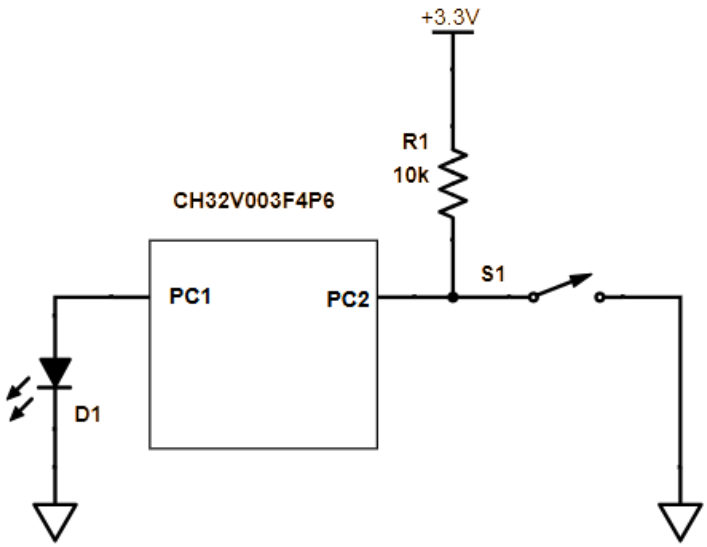
We can apply the debouncing principles learned earlier to implement a push-on / push-off button. Instead of reacting immediately to every raw button transition, the input is filtered so that only a single, clean press is detected. Each valid press toggles the internal state of the software, turning the output ON with one press and OFF with the next.

This approach eliminates false triggers caused by mechanical bounce and allows a momentary pushbutton to behave like a latching switch. The result is a stable, predictable control input that changes state only once per deliberate button press.

```
/*****************************************************************************
*Includes and defines
*****************************************************************************/
#include "debug.h"

void initGPIO(void);
uint8_t debounceButton(GPIO_TypeDef* port, uint16_t pin);

/*****************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*****************************************************************************/
void initMain(void)
{
 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
 SystemCoreClockUpdate();
 Delay_Init();
}


/*****************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*****************************************************************************/
int main(void)
{
initMain();
initGPIO();

while(1)
{
 if (debounceButton(GPIOC, GPIO_Pin_2))
 {
  // Toggle LED on PC1
  if (GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_1))
    GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_RESET);
  else
    GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_SET);
   }
 }
}
```

```c
/*****************************************************************************
* Function: void initGPIO()
* Returns: Nothing
* Description: Contains initializations for GPIO pins
* Usage: initGPIO()
*****************************************************************************/
void initGPIO(void)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

  // PC1 as output
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  // PC2 as input with pull-up
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_Init(GPIOC, &GPIO_InitStructure);
}

/*****************************************************************************
* Function: uint8_t debounceButton(GPIO_TypeDef* port, uint16_t pin)
* Returns: Debouce Output
* Description: Debounce Helper
* Usage: debounceButton(Port, Pin)
*****************************************************************************/
uint8_t debounceButton(GPIO_TypeDef* port, uint16_t pin)
{
  if (GPIO_ReadInputDataBit(port, pin) == 0) // Button pressed (active-low)
  {
    Delay_Ms(20); // Wait 20ms
    if (GPIO_ReadInputDataBit(port, pin) == 0) // Still pressed?
    {
      while (GPIO_ReadInputDataBit(port, pin) == 0); // Wait until released
      Delay_Ms(20); // Confirm release
      return 1;
    }
  }
   return 0;
}
```

Lab #20: Double Click Detection



We can also implement double-click–style detection using timing logic, building on the same principles used for debouncing. Instead of responding immediately to a button press, the system measures how long the button is pressed or how closely two presses occur in time.

In general, double-click detection, the controller notes the time of the first valid press and waits for a second press within a defined window (for example, 300–500 ms). If the second press occurs within that window, the action is treated as a double-click; otherwise, the first press is handled as a normal single-click. This allows multiple actions to be triggered using a single button.

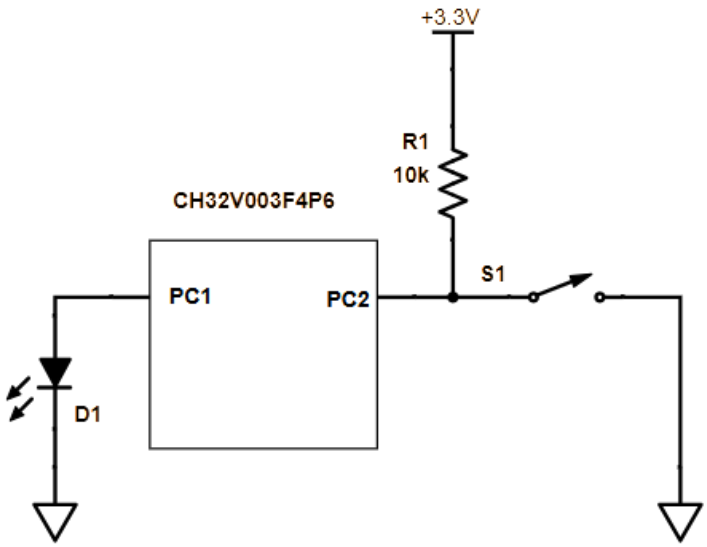
In our specific implementation, a short press (less than 1 second) does nothing. If the button is held for more than 1 second, the LED toggles. While the button remains held, no additional triggers occur. Once the button is released and held again for more than a second, a new toggle is generated, ensuring clean and predictable behavior.

```c
/*****************************************************************************
*Includes and defines
*****************************************************************************/
#include "debug.h"

void initGPIO(void);
uint8_t debounceButton(GPIO_TypeDef* port, uint16_t pin);

/*****************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*****************************************************************************/
void initMain(void)
{
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
   SystemCoreClockUpdate();
   Delay_Init();
}

/*****************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*****************************************************************************/
int main(void)
{
   uint8_t clickCount = 0;
   uint32_t timer = 0;

    initMain();
    initGPIO();

    while (1)
   {
      if (debounceButton(GPIOC, GPIO_Pin_2))
     {
        if (clickCount == 0)
        {
            clickCount = 1;
            timer = 0; // start timing window
         }
      else if (clickCount == 1 && timer < 400)
     {
       // Double click detected
       // Toggle LED on PC1
```

```
      if (GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_1))
        GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_RESET);
      else
        GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_SET);

     clickCount = 0;
     timer = 0;
   }
}

// Timing window for double-click
if (clickCount == 1)
{
  Delay_Ms(10);
  timer += 10;
   if (timer >= 400)
  {
    // Too slow → reset
   clickCount = 0;
   timer = 0;
  }
 }
}

/*******************************************************************************
* Function: void initGPIO(void)
* Returns: Nothing
* Description: Sets up PC1 as output and PC2 as input
* Usage: initGPIO()
*******************************************************************************/
void initGPIO(void)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

   // PC1 as output
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
   GPIO_Init(GPIOC, &GPIO_InitStructure);

  // PC2 as input with pull-up
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
  GPIO_Init(GPIOC, &GPIO_InitStructure);
```

```
}

/*****************************************************************************
* Function: uint8_t debounceButton(GPIO_TypeDef* port, uint16_t pin)
* Returns: Nothing
* Description: Allows us to perform software debouncing on a pin
* Usage: debounceButton(GPIOC, GPIO_Pin_2)
*****************************************************************************/
uint8_t debounceButton(GPIO_TypeDef* port, uint16_t pin)
{
  if (GPIO_ReadInputDataBit(port, pin) == 0) // Button pressed (active-low)
  {
    Delay_Ms(20); // Wait 20ms
    if (GPIO_ReadInputDataBit(port, pin) == 0) // Still pressed?
    {
      while (GPIO_ReadInputDataBit(port, pin) == 0); // Wait until released
      Delay_Ms(20); // Confirm release
      return 1;
    }
  }
  return 0;
}
```

Lab #21: Long-Press Detection



Long-press detection builds on the same timing concepts used for single-click and double-click handling, but instead of counting how many presses occur in a time window, it focuses on how long the button is held down continuously. This interaction is widely used in embedded systems and consumer electronics to provide additional functionality without adding extra buttons.

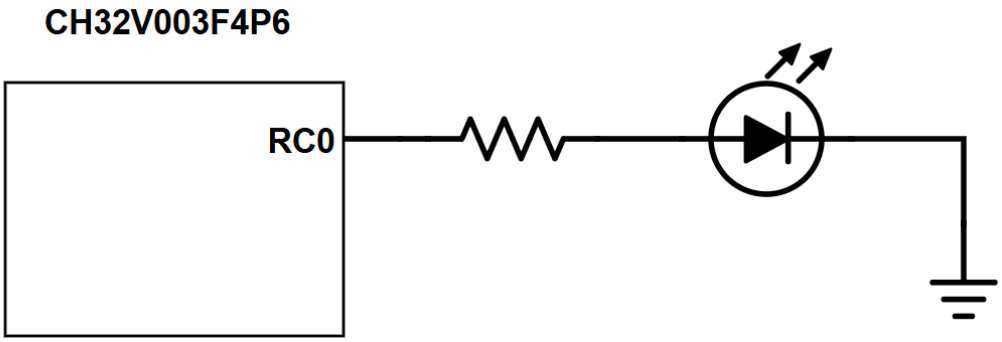
In everyday devices, a long press often triggers a different action than a quick tap. For example, holding a key on a mobile phone may display alternate characters, while holding a reset button on an embedded system may initiate a factory reset or special boot mode. This makes long-press detection a powerful and space-efficient input technique.

From an implementation standpoint, the microcontroller monitors the input pin over time after a valid (debounced) press is detected. A timer is started when the button is pressed, and if the button remains active beyond a defined threshold, the press is classified as a long press and the corresponding action is executed.

```c
/*******************************************************************************
*Includes and defines
*******************************************************************************/
#include "debug.h"

void initGPIO(void);

/*******************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*******************************************************************************/
void initMain(void)
{
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  SystemCoreClockUpdate();
  Delay_Init();
}

/*******************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*******************************************************************************/
int main(void)
{
  uint16_t holdTime = 0;
  uint8_t longPressDetected = 0;

   initMain();
   initGPIO();

while (1)
{
  // Button is being held down
  if (GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_2) == 0)
  {
    Delay_Ms(10); // Small interval
    holdTime += 10;

    // Check for long press threshold
    if (holdTime >= 1000 && !longPressDetected)
    {
      // Toggle LED on long press
      if (GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_1))
          GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_RESET);
```

```
      else
        GPIO_WriteBit(GPIOC, GPIO_Pin_1, Bit_SET);

       longPressDetected = 1; // Prevent retrigger while holding
     }
   }
   else
   {
     // Button released
     holdTime = 0;
     longPressDetected = 0;
    }
  }
}


/*******************************************************************************
* Function: void initGPIO(void)
* Returns: Nothing
* Description: Sets up PC1 as output and PC2 as input
* Usage: initGPIO()
*******************************************************************************/
void initGPIO(void)
{
   GPIO_InitTypeDef GPIO_InitStructure = {0};

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

   // PC1 as output
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
   GPIO_Init(GPIOC, &GPIO_InitStructure);

    // PC2 as input with pull-up
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
   GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

Lab #22: Interrupts (attachInterrupt())



One of the best use cases for timers you will meet over and over again will be to use a timer triggered interrupt. However before we look at timer triggered interrupt we should talk a bit about interrupts in general. In earlier chapters we introduced GPIO and polling methods, where the CPU continuously checks the state of an input pin inside a loop. While this approach is simple, it is highly inefficient. The processor wastes cycles "asking" if something has happened, even when nothing is changing. Interrupts solve this problem by inverting the control model. Instead of the CPU monitoring every possible event, peripherals or external signals can raise an interrupt request (IRQ), which causes the CPU to pause its current task and immediately service the event. This allows the microcontroller to respond quickly to asynchronous events without wasting valuable processing time.

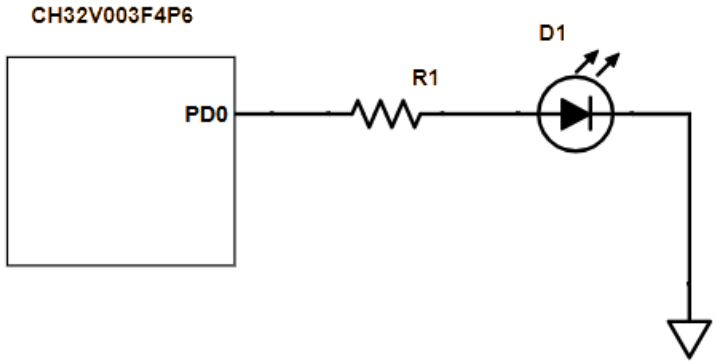
Unlike older 8-bit MCUs that often had a single shared interrupt vector (requiring the ISR to manually identify the source), the CH32V003 benefits from the Nested Vectored Interrupt Controller (NVIC). The NVIC provides multiple dedicated vectors, each associated with a specific peripheral or external source, such as timers, EXTI lines, USART, I^2C, or SPI. This means your ISR can be written specifically for one event without additional software overhead to determine its origin. The NVIC also supports priority grouping, allowing critical interrupts (for example, a safety shutoff signal) to preempt less urgent ones (such as a periodic status update).

```c
/*********************************************************************************
*Includes and defines
*********************************************************************************/
#include "debug.h"
#include <stdbool.h>

// LED defines
#define LED_GPIO_PIN GPIO_Pin_1
#define LED_GPIO_PORT GPIOC
#define LED_GPIO_RCC RCC_APB2Periph_GPIOC
#define LED_FREQ 2ul // 2Hz = 500ms
#define COUNTER_MAX 65536ul

volatile bool toggleLED = false;
void initGPIO(void);
void initTimer2(void);

/*********************************************************************************
* Function: void TIM2_IRQHandler(void)
* Returns: Nothing
* Description: Interrupt Handler
* Usage: __attribute__((interrupt("WCH-Interrupt-fast")))
* void TIM2_IRQHandler(void)
*********************************************************************************/
__attribute__((interrupt("WCH-Interrupt-fast")))
void TIM2_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        toggleLED = true;
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
    }
}

/*********************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*********************************************************************************/
void initMain()
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
    SystemCoreClockUpdate();
   Delay_Init();
}
```

```
/*******************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*******************************************************************************/
int main(void)
{
  initMain();
  initGPIO();
  initTimer2();

  BitAction ledState = Bit_RESET;

while (1)
{
   if (toggleLED)
   {
     ledState = (ledState == Bit_SET) ? Bit_RESET : Bit_SET;
     GPIO_WriteBit(LED_GPIO_PORT, LED_GPIO_PIN, ledState);
     toggleLED = false;
   }
 }
}

/*******************************************************************************
* Function: void initGPIO(void)
* Returns: Nothing
* Description: Sets up PC1 as output
* Usage: initGPIO()
*******************************************************************************/
void initGPIO(void)
{
   GPIO_InitTypeDef GPIO_InitStructure = {0};
   RCC_APB2PeriphClockCmd(LED_GPIO_RCC, ENABLE);

   GPIO_InitStructure.GPIO_Pin = LED_GPIO_PIN;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
   GPIO_Init(LED_GPIO_PORT, &GPIO_InitStructure);
}


/*******************************************************************************
* Function: void initTimer2()
* Returns: Nothing
* Description: Initialize TIM2 to trigger ever 500ms (2Hz)
* Usage: initTimer2()
```

```
*****************************************************************************/
void initTimer2(void)
{
  RCC_ClocksTypeDef clocks;
  RCC_GetClocksFreq(&clocks);

   uint16_t prescalerValue = 1;
   uint32_t freqRatio = clocks.PCLK1_Frequency / LED_FREQ;

   if (freqRatio > COUNTER_MAX)
   {
       prescalerValue = freqRatio / COUNTER_MAX;
       if (clocks.PCLK1_Frequency % prescalerValue)
       {
          prescalerValue++;
       }
        freqRatio /= prescalerValue;
     }

   prescalerValue--;
   uint16_t autoReloadValue = freqRatio - 1;

   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

  TIM_TimeBaseInitTypeDef timerInit = {0};
  timerInit.TIM_Period = autoReloadValue;
  timerInit.TIM_Prescaler = prescalerValue;
  timerInit.TIM_ClockDivision = TIM_CKD_DIV1;
  timerInit.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM2, &timerInit);

  NVIC_InitTypeDef nvicInit = {0};
  nvicInit.NVIC_IRQChannel = TIM2_IRQn;
  nvicInit.NVIC_IRQChannelPreemptionPriority = 1;
  nvicInit.NVIC_IRQChannelSubPriority = 0;
  nvicInit.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&nvicInit);

  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
  TIM_Cmd(TIM2, ENABLE);
}
```

Lab #23: Systick Timer (millis())



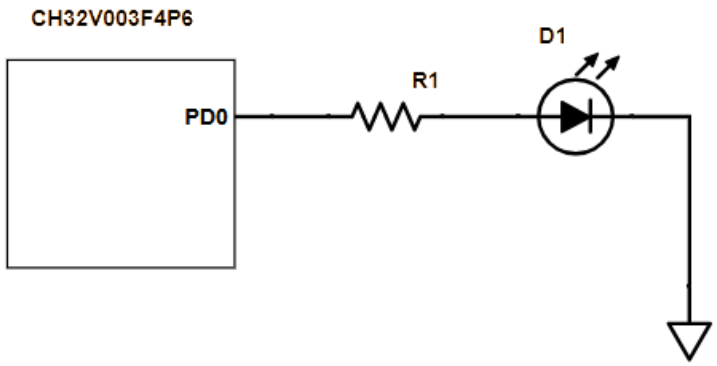
One of the best use cases for timers you will meet over and over again will be to use a timer triggered interrupt. However before we look at timer triggered interrupt we should talk a bit about interrupts in general. In earlier chapters we introduced GPIO and polling methods, where the CPU continuously checks the state of an input pin inside a loop. While this approach is simple, it is highly inefficient. The processor wastes cycles "asking" if something has happened, even when nothing is changing. Interrupts solve this problem by inverting the control model. Instead of the CPU monitoring every possible event, peripherals or external signals can raise an interrupt request (IRQ), which causes the CPU to pause its current task and immediately service the event. This allows the microcontroller to respond quickly to asynchronous events without wasting valuable processing time.

Unlike older 8-bit MCUs that often had a single shared interrupt vector (requiring the ISR to manually identify the source), the CH32V003 benefits from the Nested Vectored Interrupt Controller (NVIC). The NVIC provides multiple dedicated vectors, each associated with a specific peripheral or external source, such as timers, EXTI lines, USART, I^2C, or SPI. This means your ISR can be written specifically for one event without additional software overhead to determine its origin. The NVIC also supports priority grouping, allowing critical interrupts (for example, a safety shutoff signal) to preempt less urgent ones (such as a periodic status update).

```c
/*********************************************************************
*Includes and defines
*********************************************************************/
#include "debug.h"

/*********************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*********************************************************************/
static void initMain(void)
{
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
   SystemCoreClockUpdate();
   Delay_Init();
}

/*********************************************************************
* Function: void initGPIO(void)
* Returns: Nothing
* Description: Sets up PD0 as LED output
* Usage: initGPIO()
*********************************************************************/
static void initGPIO(void)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_Init(GPIOD, &GPIO_InitStructure);

  // Start with LED off (active-low wiring)
  GPIO_WriteBit(GPIOD, GPIO_Pin_0, Bit_SET);
}

/*********************************************************************
* Function: static void initSysTick_1Hz(void)
* Returns: Nothing
* Description: Initialize SysTick to 1 Hz
* Usage: initSysTick_1Hz()
*********************************************************************/
static void initSysTick_1Hz(void)
{
```

```
  NVIC_EnableIRQ(SysTick_IRQn);

  SysTick->SR &= ~(1U << 0); // Clear any pending flag
  SysTick->CMP = (SystemCoreClock/1) - 1; // 1 Hz tick (syscoreclock/10) for 10 Hz for example
  SysTick->CNT = 0; // Reset counter
  SysTick->CTLR = 0xF; // Enable counter + interrupt
}

/*********************************************************************
* Function: void SysTick_Handler(void)
* Returns: Nothing
* Description: SysTick ISR
* Usage: void SysTick_Handler(void) __attribute__((interrupt("WCH-Interrupt-fast")));
* void SysTick_Handler(void)
*********************************************************************/
void SysTick_Handler(void) __attribute__((interrupt("WCH-Interrupt-fast")));
void SysTick_Handler(void)
{
  // Toggle PD0 (active-low LED)
  if (GPIO_ReadOutputDataBit(GPIOD, GPIO_Pin_0))
   GPIO_WriteBit(GPIOD, GPIO_Pin_0, Bit_RESET);
  else
    GPIO_WriteBit(GPIOD, GPIO_Pin_0, Bit_SET);
    SysTick->SR = 0; // Clear SysTick interrupt flag
}

/*********************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*********************************************************************/
int main(void)
{
  initMain();
  initGPIO();
  initSysTick_1Hz();

   while (1)
  {
     // Idle loop  LED toggle happens in SysTick interrupt
  }
}
```

Lab #24: Watchdog (wdt_enable())



One of the best use cases for timers you will meet over and over again will be to use a timer triggered interrupt. However before we look at timer triggered interrupt we should talk a bit about interrupts in general. In earlier chapters we introduced GPIO and polling methods, where the CPU continuously checks the state of an input pin inside a loop. While this approach is simple, it is highly inefficient. The processor wastes cycles "asking" if something has happened, even when nothing is changing. Interrupts solve this problem by inverting the control model. Instead of the CPU monitoring every possible event, peripherals or external signals can raise an interrupt request (IRQ), which causes the CPU to pause its current task and immediately service the event. This allows the microcontroller to respond quickly to asynchronous events without wasting valuable processing time.

Unlike older 8-bit MCUs that often had a single shared interrupt vector (requiring the ISR to manually identify the source), the CH32V003 benefits from the Nested Vectored Interrupt Controller (NVIC). The NVIC provides multiple dedicated vectors, each associated with a specific peripheral or external source, such as timers, EXTI lines, USART, I^2C, or SPI. This means your ISR can be written specifically for one event without additional software overhead to determine its origin. The NVIC also supports priority grouping, allowing critical interrupts (for example, a safety shutoff signal) to preempt less urgent ones (such as a periodic status update).

```c
/*******************************************************************************
*Includes and defines
*******************************************************************************/
#include "debug.h"

#define FEED_WATCHDOG 0 // 1 = keep alive; 0 = demonstrate reset after timeout

// ===== LED helpers (active-low) =====
static inline void led_on(void) { GPIO_WriteBit(GPIOD, GPIO_Pin_0, Bit_RESET); }
static inline void led_off(void) { GPIO_WriteBit(GPIOD, GPIO_Pin_0, Bit_SET); }
static inline void led_toggle(void)
{
    if (GPIO_ReadOutputDataBit(GPIOD, GPIO_Pin_0)) led_on(); else led_off();
}

/*******************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*******************************************************************************/
static void initMain(void)
{
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  SystemCoreClockUpdate();
  Delay_Init();
}

/*******************************************************************************
* Function: static void initGPIO(void)
* Returns: Nothing
* Description: Sets up PD0 as output
* Usage: initGPIO()
*******************************************************************************/
static void initGPIO(void)
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_Init(GPIOD, &GPIO_InitStructure);

  led_off(); // start LED off (active-low)
}
```

```
/*******************************************************************************
* Function: static void boot_blink(void)
* Returns: Nothing
* Description: Quick triple-blink on LED so we can see resets
* Usage: boot  blink()
*******************************************************************************/
static void boot_blink(void)
{
  // Quick triple-blink so you can see resets
  for (int i = 0; i < 3; ++i) { led_on(); Delay_Ms(80); led_off(); Delay_Ms(80); }
}

/*******************************************************************************
* Function: static void iwdg  init(void)
* Returns: Nothing
* Description: Independent watchdog timer init 128 prescaler, 4000 (4s) reload
* Usage: iwdg_init()
*******************************************************************************/
static void iwdg  init(void)
{
  IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
  IWDG_SetPrescaler(IWDG_Prescaler_128);
  IWDG_SetReload(4000);
  IWDG_ReloadCounter();
  IWDG_Enable();
}

/*******************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*******************************************************************************/
int main(void)
{
  initMain();
  initGPIO();
  boot_blink();
  iwdg_init();

  uint32_t ms = 0;

while (1)
{
  led_toggle();
  Delay_Ms(200);
  ms += 200;
```

```
  #if FEED_WATCHDOG
    // Feed often enough (< timeout). Here: every 400 ms.
   if (ms >= 400) {
      IWDG_ReloadCounter();
     ms = 0;
   }
   #endif
    // If FEED_WATCHDOG == 0, we never reload → device will reset after ~4 s.
     }
}
```

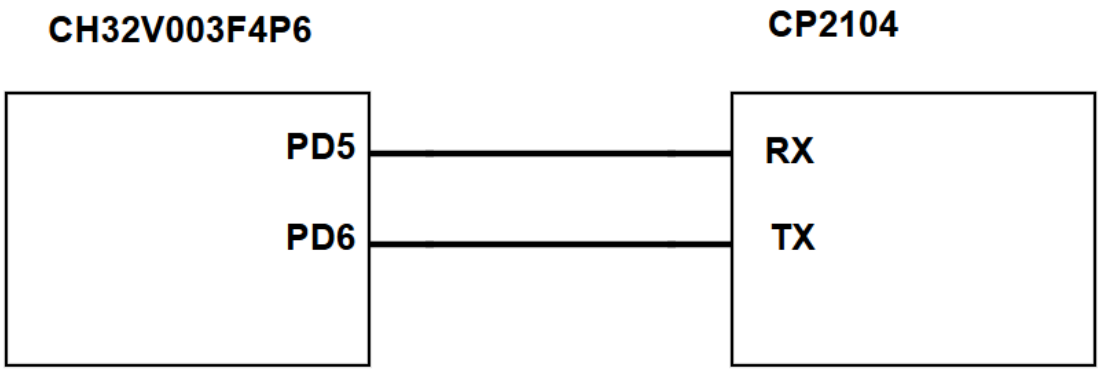
Serial and Parallel Communication

When working with microcontroller-based systems, one of the most important skills you will need to master is communication. A single microcontroller can blink LEDs, spin a motor, or run simple programs all on its own, but real-world applications often demand more. You might need to gather readings from a temperature sensor, send data to a display, exchange information with another microcontroller, or even connect to a computer for logging and control. To make all of this possible, microcontrollers rely on communication protocols, which are structured methods that define how data is packaged, transmitted, and received. This is the heart of communication systems on these devices. Communication is a valuable tool, and just as in your daily life it is near impossible to live without having some sort of communication, in our internet connected era, its almost just as impossible to build modern embedded devices without involving some sort of communication protocol.

Before you can begin programming communication routines, it is essential to understand both the protocol itself and the device you intend to interface with. Each protocol defines rules for data exchange, but every device also has its own requirements and quirks. For example, a digital temperature sensor may expect data in a particular byte format, while a display module might require initialization commands before it can show text. Mastering embedded communication therefore means not just knowing how to configure your microcontroller, but also how to "speak the language" of each peripheral you wish to interface with. This is important because each peripheral or sensor you interface with may perform the same or similar function, but may be made by entirely different manufacturers. Think about the temperature sensors we are talking about. How many different temperature sensors are there? how many manufacturers? Imagine if we did not have standard methods of communication. Ask any embedded developer that has to deal with proprietary communication protocols how frustrating that experience is, and you'll appreciate why we need a standard for common protocols and why it's so important to interface with them.

In this chapter, we will focus on three of the most widely used communication protocols in embedded systems development: UART, SPI, and I^2C. These interfaces are supported by nearly every modern microcontroller as there are usually hardware blocks to cater to using these protocols, this includes the CH32V003. The nice thing about these communication protocols as we will see later is that even if the device we are using lacks hardware or there is a shortage of modules we can interface with, then we can implement the solutions in software.

Lab #25: UART (Serial.begin(()))



One of the best use cases for timers you will meet over and over again will be to use a timer triggered interrupt. However before we look at timer triggered interrupt we should talk a bit about interrupts in general. In earlier chapters we introduced GPIO and polling methods, where the CPU continuously checks the state of an input pin inside a loop. While this approach is simple, it is highly inefficient. The processor wastes cycles "asking" if something has happened, even when nothing is changing. Interrupts solve this problem by inverting the control model. Instead of the CPU monitoring every possible event, peripherals or external signals can raise an interrupt request (IRQ), which causes the CPU to pause its current task and immediately service the event. This allows the microcontroller to respond quickly to asynchronous events without wasting valuable processing time.

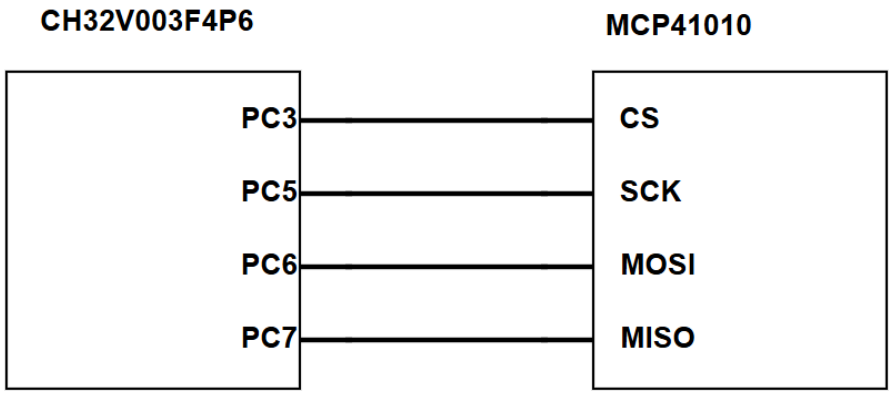
Unlike older 8-bit MCUs that often had a single shared interrupt vector (requiring the ISR to manually identify the source), the CH32V003 benefits from the Nested Vectored Interrupt Controller (NVIC). The NVIC provides multiple dedicated vectors, each associated with a specific peripheral or external source, such as timers, EXTI lines, USART, I^2C, or SPI. This means your ISR can be written specifically for one event without additional software overhead to determine its origin. The NVIC also supports priority grouping, allowing critical interrupts (for example, a safety shutoff signal) to preempt less urgent ones (such as a periodic status update).

```
/*****************************************************************************
*Includes and defines
*****************************************************************************/
#include "debug.h"
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define RX_BUF_SIZE 32

static int is_number(const char *s);
static void USART1_Init(void);


/*****************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*****************************************************************************/
static void initMain(void) {
 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
 SystemCoreClockUpdate();
 Delay_Init();
}


/*****************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*****************************************************************************/
int main(void) {
  initMain();
  USART1_Init();

  char buf[RX_BUF_SIZE];
  uint8_t idx = 0;

   while (1) {
    if (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET) {
      char c = USART_ReceiveData(USART1) & 0xFF;

     if (c == '\r' || c == '\n') {
       buf[idx] = '\0';
       idx = 0;
```

```
      if (is_number(buf)) {
        long val = strtol(buf, NULL, 10);
        char out[16];
        snprintf(out, sizeof(out), "%ld\r\n", val + 1);
        for (char *p = out; *p; p++) {
          USART_SendData(USART1, (uint8_t)*p);
          while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
       }
      } else if (buf[0] != '\0') {
        const char reply[] = "yes?\r\n";
        for (const char *p = reply; *p; p++) {
       USART_SendData(USART1, (uint8_t)*p);
        while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
       }
     }
    } else {
    if (idx < RX_BUF_SIZE - 1) buf[idx++] = c;
      }
     }
   }
}

/*******************************************************************************
* Function: static void USART1_Init(void)
* Returns: Nothing
* Description: Initalizes the UART module
* Usage: USART1_Init()
*******************************************************************************/
static void USART1_Init(void) {
    GPIO_InitTypeDef GPIO_InitStructure = {0};
    USART_InitTypeDef USART_InitStructure = {0};

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_USART1, ENABLE);

   // TX = PD5 (AF push-pull), RX = PD6 (floating input)
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_30MHz;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
   GPIO_Init(GPIOD, &GPIO_InitStructure);
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
   GPIO_Init(GPIOD, &GPIO_InitStructure);

   USART_InitStructure.USART_BaudRate = 115200;
   USART_InitStructure.USART_WordLength = USART_WordLength_8b;
   USART_InitStructure.USART_StopBits = USART_StopBits_1;
   USART_InitStructure.USART_Parity = USART_Parity_No;
```

```
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
  USART_Init(USART1, &USART_InitStructure);
  USART_Cmd(USART1, ENABLE);
}

/*******************************************************************************
* Function: static int is_number(const char *s)
*
* Returns: If is number or not
*
* Description: Checks if character is a number
*
* Usage: if (is_number(buf))
*******************************************************************************/
static int is_number(const char *s) {
  if (*s == '\0') return 0;
    while (*s) {
      if (!isdigit((unsigned char)*s)) return 0;
      s++;
    }
    return 1;
}
```

Lab #26: SPI (SPI.begin())



Unlike UART, which is asynchronous, SPI is a synchronous serial communication protocol. This means that the communication between devices is coordinated by a shared clock signal rather than relying on precisely matched baud rates and start/stop framing. While we know that there is a synchronous version of UART (USART) that can use a clock, the majority of the time we use the protocol will be asynchronously. The presence of a clock at all times is what makes SPI both simpler and faster in many respects. Because one device (we call the master) controls the timing, the other device (we call the slave) does not need to guess when each bit is arriving, it simply samples data in step with the master's clock. This results in extremely reliable and high-speed communication, often much faster than what you can achieve with UART.

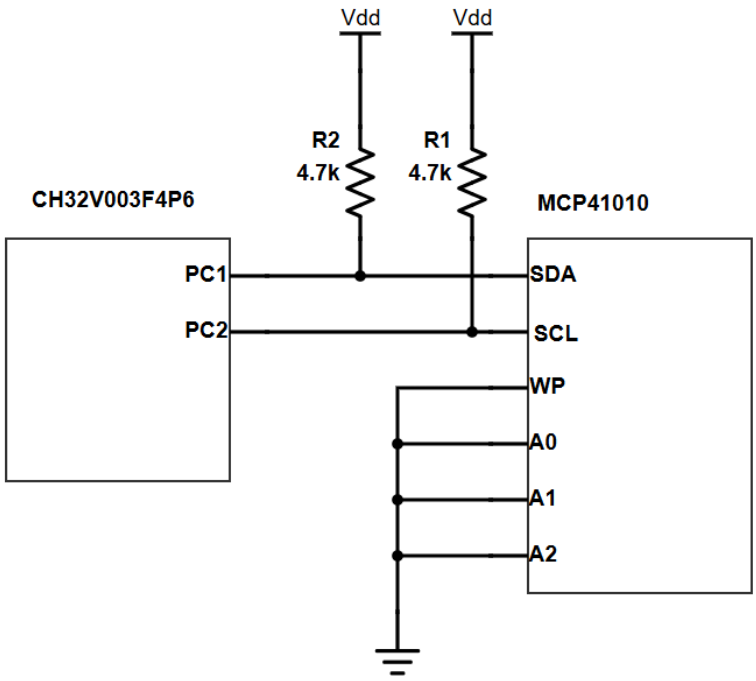
At its core, SPI uses four signal lines. The first is the serial clock line, called SCK or SCLK, which carries the clock signal generated by the master device. The second is the master's data output line, called SDO or MOSI (Master Out Slave In). The third is the master's data input line, called SDI or MISO (Master In Slave Out). Finally, there is the chip select line, often labeled CS, SS (Slave Select), or sometimes CE (Chip Enable), which is used to enable a particular slave device. These four lines form the basis of most SPI systems, although in the most basic configuration, you can technically use only three if you do not need two-way communication. Once you have this basic setup, the way the protocol is designed you will have no issue adding additional devices.

```c
/*********************************************************************************
*Includes and defines
*********************************************************************************/
#include "ch32v00x.h"
#include "ch32v00x_gpio.h"
#include "ch32v00x_rcc.h"
#include "ch32v00x_spi.h"
#include "debug.h"

#define MCP41010_CMD_WRITE 0x11
#define MCP_CS_PORT GPIOC
#define MCP_CS_PIN GPIO_Pin_3

static void SPI1_Init_Master(void);
static void MCP41010_Init(void);
static void MCP41010_SetWiper(uint8_t value);
static uint8_t spi1_transfer(uint8_t data);

// inline functions for toggling CS pins
static inline void cs_low(void) { GPIO_ResetBits(MCP_CS_PORT, MCP_CS_PIN); }
static inline void cs_high(void) { GPIO_SetBits(MCP_CS_PORT, MCP_CS_PIN); }

/*********************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*********************************************************************************/
int main(void)
{
  SystemCoreClockUpdate();
  Delay_Init();

  SPI1_Init_Master();
  MCP41010_Init();

   while (1)
  {
     for (uint8_t i = 0; i < 255; i++) { // 0..254
        MCP41010_SetWiper(i);
       Delay_Ms(10);
      }
   }
}

/*********************************************************************************
* Function: static void SPI1_Init_Master(void)
* Usage: PI1_Init_Master()
```

```
*****************************************************************************/
static void SPI1_Init_Master(void)
{
   GPIO_InitTypeDef gpio = {0};
   SPI_InitTypeDef spi = {0};

  /* Enable clocks for GPIOC and SPI1 */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC | RCC_APB2Periph_SPI1, ENABLE);

  /* CS (PC3) as push-pull output, high speed, idle high */
  gpio.GPIO_Pin = MCP_CS_PIN;
  gpio.GPIO_Mode = GPIO_Mode_Out_PP;
  gpio.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(MCP_CS_PORT, &gpio);
  GPIO_SetBits(MCP_CS_PORT, MCP_CS_PIN);

  /* SPI pins: PC5=SCK (AF_PP), PC6=MOSI (AF_PP), PC7=MISO (floating input) */
  gpio.GPIO_Pin = GPIO_Pin_5;
  gpio.GPIO_Mode = GPIO_Mode_AF_PP;
  gpio.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOC, &gpio);

  gpio.GPIO_Pin = GPIO_Pin_6;
  gpio.GPIO_Mode = GPIO_Mode_AF_PP;
  gpio.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOC, &gpio);

  gpio.GPIO_Pin = GPIO_Pin_7;
  gpio.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOC, &gpio);

  /* SPI configuration:
  - Master, 8-bit
  - CPOL=0, CPHA=1 (data captured on the second edge) per your original settings (CPHA  2Edge)
  - Software NSS
  - Baud prescaler: start slow (e.g., /256) and increase after validation
  - MSB first
  */
  spi.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
  spi.SPI_Mode = SPI_Mode_Master;
  spi.SPI_DataSize = SPI_DataSize_8b;
  spi.SPI_CPOL = SPI_CPOL_Low;
  spi.SPI_CPHA = SPI_CPHA_1Edge;
  spi.SPI_NSS = SPI_NSS_Soft;
  spi.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256; // safe; MCP41010 allows up to ~10MHz, tune later
  spi.SPI_FirstBit = SPI_FirstBit_MSB;
  spi.SPI_CRCPolynomial = 7;
```

```
 SPI_Init(SPI1, &spi);
 SPI_Cmd(SPI1, ENABLE);
}

/*****************************************************************************
* Function: static void MCP41010_Init(void)
* Usage: MCP41010_Init()
*****************************************************************************/
static void MCP41010_Init(void)
{
  // CS pin already configured in SPI1_Init_Master. Keep CS high.
  GPIO_SetBits(MCP_CS_PORT, MCP_CS_PIN);
}

/*****************************************************************************
* Function: static uint8_t spi1_transfer(uint8_t data)
* Returns: Byte recieved on the SPI bus
* Description: Transfer a byte of data via SPI
* Usage: (void)spi1_transfer(value)
*****************************************************************************/
static uint8_t spi1_transfer(uint8_t data)
{
 /* Wait TXE, send byte */
 while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET);
 SPI_I2S_SendData(SPI1, data);

 /* Wait RXNE, read received byte */
 while (SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_RXNE) == RESET);
 return (uint8_t)SPI_I2S_ReceiveData(SPI1);
}


/*****************************************************************************
* Function: static void MCP41010_SetWiper(uint8_t value)
* Returns: Nothing
* Description: Set the position of the MCP41010
* Usage: MCP41010_SetWiper(i)
*****************************************************************************/
static void MCP41010_SetWiper(uint8_t value)
{
  cs_low();
  (void)spi1_transfer(MCP41010_CMD_WRITE); // command
  (void)spi1_transfer(value); // data
  cs_high();
}
```

Lab #27: I2C (Wire.begin())



Like SPI, I²C is a synchronous serial protocol, but it is quite different in how it operates and what it is designed for. While SPI emphasizes speed and simplicity, I²C is built around flexibility and simplicity of wiring. It is designed to allow a controller (like SPI often called the master) to communicate with many target devices (also like SPI often called slaves) over just two wires. This makes it extremely popular for connecting low-to-moderate bandwidth peripherals such as sensors, EEPROMs, real-time clocks, codecs, and other support devices inside embedded systems.

What makes I²C extremely popular is that it uses only two signal lines, called SCL (serial clock) and SDA (serial data). Both lines are "open-drain" (or "open-collector" in bipolar designs), which means that devices on the bus cannot drive the lines high, they can only pull them low. Pull-up resistors are used to bring the lines to a logic high level when no device is pulling them low. Because of this wired-AND arrangement, many devices can share the same two wires without conflict. Any device can pull the line low, and the resulting level on the bus will always reflect the lowest value driven by any participant. This property is what allows I²C to support multiple devices and even multiple controllers on the same bus without additional wiring.

```c
/*******************************************************************************
*Includes and defines
*******************************************************************************/
#include "ch32v00x.h"
#include "ch32v00x_rcc.h"
#include "ch32v00x_gpio.h"
#include "ch32v00x_i2c.h"
#include "debug.h"
#include <stdio.h>

static inline uint8_t eeprom_ctrl(uint16_t word_addr);
static int wait_event(uint32_t evt, uint32_t ms);
static int wait_flag_set(FlagStatus (*fn)(I2C_TypeDef*, uint32_t),
I2C_TypeDef *I2Cx, uint32_t flag, uint32_t ms);
static int wait_flag_clear(FlagStatus (*fn)(I2C_TypeDef*, uint32_t),
I2C_TypeDef *I2Cx, uint32_t flag, uint32_t ms);
static void i2c_bus_soft_recover(void);
static void I2C1_Init_CH32(uint32_t clock_hz);
static int i2c_start(void);
static int i2c_restart(void);
static void i2c_stop(void);
static int addr_tx(uint8_t ctrl8);
static int addr_rx(uint8_t ctrl8);
static int send_byte(uint8_t b);
static int eeprom_write_byte(uint16_t word_addr, uint8_t data);
static int eeprom_read_byte(uint16_t word_addr, uint8_t *out);

/*******************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*******************************************************************************/
void initmain()
{
  SystemCoreClockUpdate();
   Delay_Init();
}

/*******************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*******************************************************************************/
int main(void)
{
   initmain();
```

```
 #if (SDI_PRINT == SDI_PR_OPEN)
  SDI_Printf_Enable();
 #else
  USART_Printf_Init(115200);
 #endif

  printf("SystemClk:%ld\r\n", SystemCoreClock);
  printf("CH32V003 I2C + 24LC16B (event-driven, 8-bit ctrl + proper RESTART)\r\n");

  I2C1_Init_CH32(400000); // start at 100 kHz
  uint16_t addr = 0;
  uint8_t wr = 8;

  while (1)
  {
    printf("Write addr=%u data=%u\r\n", addr, wr);
    if (!eeprom_write_byte(addr, wr)) {
      printf("Write timeout @%u\r\n", addr);
    }

    uint8_t rd = 0xFF;
    if (!eeprom_read_byte(addr, &rd)) {
       printf("Read timeout @%u\r\n\r\n", addr);
    } else {
        printf("Read addr=%u data=%u\r\n\r\n", addr, rd);
     }

     wr += 2;
     addr = (addr + 1) & 0x07FF; // 0..2047
     Delay_Ms(1000);
  }
}

/*********************************************************************************
* Function: static inline uint8_t eeprom_ctrl(uint16_t word_addr)
* Returns: Nothing
* Description: block = A10..A8 (0..7), base = 0xA0
* For reads we still pass the same ctrl byte to I2C_Send7bitAddress(),
* but with I2C_Direction_Receiver (the lib sets R/W for us)
* Usage: uint8_t ctrl8 = eeprom_ctrl(word_addr);
*********************************************************************************/
static inline uint8_t eeprom_ctrl(uint16_t word_addr)
{
  uint8_t block = (word_addr >> 8) & 0x07; // A10..A8
  return (uint8_t)(0xA0 | (block << 1)); // 0xA0,0xA2,...,0xAE
}
```

```
/*****************************************************************************
* Function: static int wait_event(uint32_t evt, uint32_t ms)
* Returns: Status of wait event
* Description: Timeout for a wait event
* Usage: PI1_Init_Master()
*****************************************************************************/
static int wait_event(uint32_t evt, uint32_t ms)
{
  while (!I2C_CheckEvent(I2C1, evt)) { Delay_Ms(1); if (!ms--) return 0; }
  return 1;
}

/*****************************************************************************
* Function: static int wait_flag_set(FlagStatus (*fn)(I2C_TypeDef*, uint32_t),
* I2C_TypeDef *I2Cx, uint32_t flag, uint32_t ms)
* Returns: Status of the wait flag
* Description: Wait flag event
* Usage: if (!wait_flag_set(I2C_GetFlagStatus, I2C1, I2C_FLAG_RXNE, 20))
*****************************************************************************/
static int wait_flag_set(FlagStatus (*fn)(I2C_TypeDef*, uint32_t),
I2C_TypeDef *I2Cx, uint32_t flag, uint32_t ms)
{
 while (fn(I2Cx, flag) == RESET) { Delay_Ms(1); if (!ms--) return 0; }
 return 1;
}

/*****************************************************************************
* Function: static int wait_flag_clear(FlagStatus (*fn)(I2C_TypeDef*, uint32_t),
* I2C_TypeDef *I2Cx, uint32_t flag, uint32_t ms)
* Returns: Status of the wait flag
* Description: Wait flag clear status
* Usage: if (!wait_flag_clear(I2C_GetFlagStatus, I2C1, I2C_FLAG_BUSY, 20))
*****************************************************************************/
static int wait_flag_clear(FlagStatus (*fn)(I2C_TypeDef*, uint32_t),
I2C_TypeDef *I2Cx, uint32_t flag, uint32_t ms)
{
  while (fn(I2Cx, flag) != RESET) { Delay_Ms(1); if (!ms--) return 0; }
  return 1;
}

/*****************************************************************************
* Function: static void i2c_bus_soft_recover(void)
* Returns: Nothing
* Description: Fucntion to help the I2C bus recover
* Usage: i2c_bus_soft_recover()
*****************************************************************************/
```

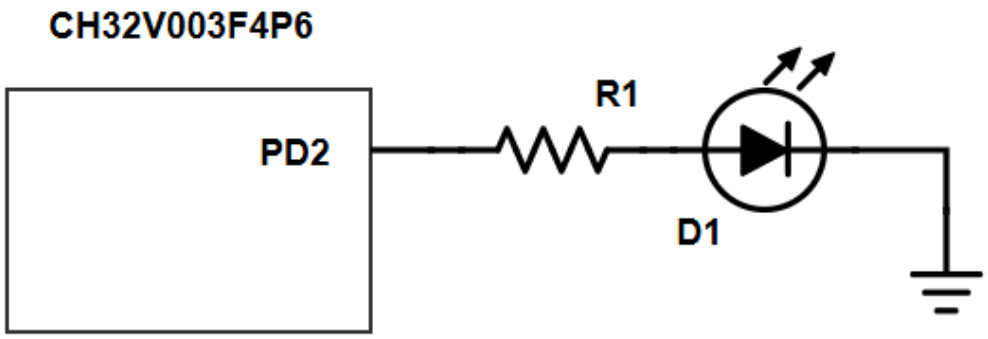
```c
static void i2c_bus_soft_recover(void)
{
   I2C_SoftwareResetCmd(I2C1, ENABLE);
   Delay_Ms(1);
   I2C_SoftwareResetCmd(I2C1, DISABLE);
}

/*******************************************************************************
* Function: static void I2C1_Init_CH32(uint32_t clock_hz)
* Returns: Nothing
* Description: Function to initialize the I2C bus
* Usage: I2C1_Init_CH32(100000)
*******************************************************************************/
static void I2C1_Init_CH32(uint32_t clock_hz)
{
  GPIO_InitTypeDef gpio = {0};
  I2C_InitTypeDef i2c = {0};

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC | RCC_APB2Periph_AFIO, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);

  /* PC2=SCL, PC1=SDA, AF-OD, 50 MHz */
 gpio.GPIO_Speed = GPIO_Speed_50MHz;
 gpio.GPIO_Mode = GPIO_Mode_AF_OD;

 gpio.GPIO_Pin = GPIO_Pin_2; GPIO_Init(GPIOC, &gpio);
 gpio.GPIO_Pin = GPIO_Pin_1; GPIO_Init(GPIOC, &gpio);

 i2c.I2C_ClockSpeed = clock_hz; // e.g. 100k
 i2c.I2C_Mode = I2C_Mode_I2C;
 i2c.I2C_DutyCycle = I2C_DutyCycle_2;
 i2c.I2C_OwnAddress1 = 0x00; // don't care
 i2c.I2C_Ack = I2C_Ack_Enable;
 i2c.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
 I2C_Init(I2C1, &i2c);
 I2C_Cmd(I2C1, ENABLE);

 /* Explicit ACK enable, same as Pallav's code */
 I2C_AcknowledgeConfig(I2C1, ENABLE);
}


/*******************************************************************************
* Function: static int i2c_start(void)
* Returns: Wait event status
* Description: START/RESTART/STOP
* Usage: i2c_start()
```

```
*****************************************************************************/
static int i2c_start(void)
{
   if (!wait_flag_clear(I2C_GetFlagStatus, I2C1, I2C_FLAG_BUSY, 20)) {
      i2c_bus_soft_recover();
    if (!wait_flag_clear(I2C_GetFlagStatus, I2C1, I2C_FLAG_BUSY, 20))
      return 0;
   }
   I2C_GenerateSTART(I2C1, ENABLE);
   return wait_event(I2C_EVENT_MASTER_MODE_SELECT, 20); // EV5
}

/*****************************************************************************
* Function: static int i2c_restart(void)
* Returns: Wait event status
* Description: Proper repeated start, don't wait for BUSY to clear
* Usage: i2c_start()
*****************************************************************************/
static int i2c_restart(void)
{
  I2C_GenerateSTART(I2C1, ENABLE);
  return wait_event(I2C_EVENT_MASTER_MODE_SELECT, 20); // EV5
}

/*****************************************************************************
* Function: static int i2c_restart(void)
* Returns: Nothing
* Description: Stops the I2C bus
* Usage: i2c_stop()
*****************************************************************************/
static void i2c_stop(void)
{
  I2C_GenerateSTOP(I2C1, ENABLE);
   (void)wait_flag_clear(I2C_GetFlagStatus, I2C1, I2C_FLAG_BUSY, 10);
}

/*****************************************************************************
* Function: static int addr_tx(uint8_t ctrl8)
* Returns: Wait event status
* Description: Address pahse using 8-bit control byte vis Send7bitAddress
* Usage: if (!addr_tx(ctrl8)) { i2c_stop(); return 0; }
*****************************************************************************/
static int addr_tx(uint8_t ctrl8)
{
   I2C_Send7bitAddress(I2C1, ctrl8, I2C_Direction_Transmitter);
   return wait_event(I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED, 20); // EV6
}
```

```c
/*******************************************************************************
* Function: static int addr_rx(uint8_t ctrl8)
* Returns: Wait event status
* Description: Address rx
* Usage: if (!addr_rx(ctrl8)) { i2c_stop(); return 0; }
*******************************************************************************/
static int addr_rx(uint8_t ctrl8)
{
   I2C_Send7bitAddress(I2C1, ctrl8, I2C_Direction_Receiver);
   return wait_event(I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED, 20); // EV6
}

/*******************************************************************************
* Function: static int send_byte(uint8_t b)
* Returns: Wait event status
* Description: Event-based daat byte helpers
* Usage: if (!send_byte(data)){ i2c_stop(); return 0; }
*******************************************************************************/
static int send_byte(uint8_t b)
{
  I2C_SendData(I2C1, b);
  return wait_event(I2C_EVENT_MASTER_BYTE_TRANSMITTED, 20); // EV8_2
}

/*******************************************************************************
* Function: static int send_byte(uint8_t b)
* Returns: Status
* Description: 24LC16B single-byte write simple tWR relay, no polling
* Usage: if (!eeprom_write_byte(addr, wr))
*******************************************************************************/
static int eeprom_write_byte(uint16_t word_addr, uint8_t data)
{
  uint8_t ctrl8 = eeprom_ctrl(word_addr);
   uint8_t mem = (uint8_t)(word_addr & 0xFF);

   /* START + ctrlW + mem + data + STOP */
  if (!i2c_start()) return 0;
  if (!addr_tx(ctrl8)) { i2c_stop(); return 0; }
  if (!send_byte(mem)) { i2c_stop(); return 0; }
  if (!send_byte(data)){ i2c_stop(); return 0; }
  i2c_stop();

  /* 24LC16B write-cycle time ~5ms (tWR). Wait slightly over and report success. */
 Delay_Ms(6);
 return 1;
}
```

```
/*****************************************************************************
* Function: static int eeprom_read_byte(uint16_t word_addr, uint8_t *out)
* Returns: Status
* Description: 24LC16B single-byte random read
* Usage: if (!eeprom_read_byte(addr, &rd))
*****************************************************************************/
static int eeprom_read_byte(uint16_t word_addr, uint8_t *out)
{
  uint8_t ctrl8 = eeprom_ctrl(word_addr);
  uint8_t mem = (uint8_t)(word_addr & 0xFF);

  /* Send memory address */
  if (!i2c_start()) return 0;
  if (!addr_tx(ctrl8)) { i2c_stop(); return 0; }
  if (!send_byte(mem)) { i2c_stop(); return 0; }

  /* Repeated START + same ctrl (Receiver dir) — use i2c_restart() */
  if (!i2c_restart()) { i2c_stop(); return 0; }
  if (!addr_rx(ctrl8)) { i2c_stop(); return 0; }

  /* Read one byte: NACK then STOP */
 I2C_AcknowledgeConfig(I2C1, DISABLE);
 if (!wait_flag_set(I2C_GetFlagStatus, I2C1, I2C_FLAG_RXNE, 20)) {
 I2C_AcknowledgeConfig(I2C1, ENABLE);
 i2c_stop();
 return 0;
}

  *out = I2C_ReceiveData(I2C1);
  I2C_GenerateSTOP(I2C1, ENABLE);
  I2C_AcknowledgeConfig(I2C1, ENABLE);
  return 1;
}
```

Lab #28: PWM (analogWrite())



We will start off by discussing Pulse Width Modulation or PWM. PWM relies on a simple principle of operation; instead of continuously varying voltage, PWM rapidly switches a signal between high (on) and low (off) states at a fixed frequency. The key variable is the duty cycle, which represents the fraction of time the signal remains high in each cycle. A 25% duty cycle means the signal is active for one quarter of the period, while a 75% duty cycle keeps it active for most of the time.

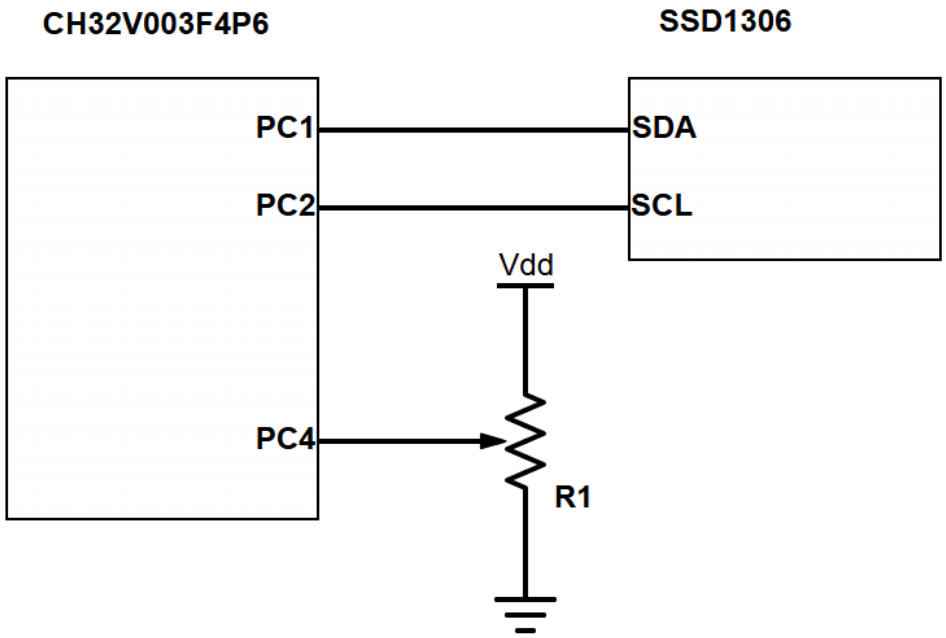
To the human eye, or to an electrical load with inherent inertia or filtering, these high-frequency pulses blend into a smooth, continuous power level. This makes PWM not only energy efficient but also ideal for applications such as dimming lights, controlling motor speed, regulating temperature, and even producing analog-like signals. PWM is important because a microcontroller device can't produce analog voltage directly, but we can fake it by operating at so much speed that the analog device doesn't even notice.

```c
/*********************************************************************************
*Includes and defines
*********************************************************************************/
#include "debug.h"

// Constants
#define PWM_MODE1 0
#define PWM_MODE2 1

// Select PWM Mode
// You can switch to PWM_MODE2 if desired
#define PWM_MODE PWM_MODE1

void initPWM_TIM1(uint16_t period, uint16_t prescaler, uint16_t duty);
void initPWM_GPIO();

/*********************************************************************************
* Function: void initMain()
* Returns: Nothing
* Description: Contains initializations for main
* Usage: initMain()
*********************************************************************************/
void initMain()
{
 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
 SystemCoreClockUpdate();
 Delay_Init();
}

/*********************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*********************************************************************************/
int main(void)
{
  uint16_t period = 2399; // ARR
  uint16_t prescaler = 0; // PSC

  initMain(); // Core system setup
  initPWM_GPIO(); // Setup PD2 for PWM
  initPWM_TIM1(period, prescaler, 1200); // PWM: 20kHz freq, 50% duty cycle

while(1)
{
  // PWM signal continues running in hardware
}
```

```
}


/*****************************************************************************
* Function: void initPWM_TIM1(uint16_t period, uint16_t prescaler, uint16_t duty)
* Returns: Nothing
* Description: Initialize Timer1 in PWM Output Mode on Channel 1
* Usage: initPWM_TIM1(period, prescaler, duty)
*****************************************************************************/
void initPWM_TIM1(uint16_t period, uint16_t prescaler, uint16_t duty)
{
  TIM_OCInitTypeDef TIM_OCInitStructure = {0};
  TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure = {0};

  // Enable clock for TIM1
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE);

 // Basic timer setup
 TIM_TimeBaseInitStructure.TIM_Period = period;
 TIM_TimeBaseInitStructure.TIM_Prescaler = prescaler;
 TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
 TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
 TIM_TimeBaseInit(TIM1, &TIM_TimeBaseInitStructure);

 #if (PWM_MODE == PWM_MODE1)
 TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
 #elif (PWM_MODE == PWM_MODE2)
 TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
 #endif

 // PWM configuration
 TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
 TIM_OCInitStructure.TIM_Pulse = duty;
 TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
 TIM_OC1Init(TIM1, &TIM_OCInitStructure);

 // Enable PWM output and start timer
 TIM_CtrlPWMOutputs(TIM1, ENABLE);
 TIM_OC1PreloadConfig(TIM1, TIM_OCPreload_Disable);
 TIM_ARRPreloadConfig(TIM1, ENABLE);
 TIM_Cmd(TIM1, ENABLE);
}

/*****************************************************************************
* Function: void initPWM_GPIO()
* Returns: Nothing
* Description: Sets up GPIO for PWM output on PD2
```

```
* Usage: initPWM_GPIO()
*****************************************************************************/
void initPWM_GPIO()
{
  GPIO_InitTypeDef GPIO_InitStructure = {0};

 // Enable clock for GPIOD
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

 // Configure PD2 as Alternate Function Push-Pull
 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
 GPIO_Init(GPIOD, &GPIO_InitStructure);
}
```

Lab #29: ADC (analogRead())



Most microcontrollers, including the CH32V series, use successive-approximation (SAR) ADCs. A SAR ADC works in two main steps. First, a sample-and-hold circuit captures the input voltage and holds it steady while the conversion takes place. Then, a binary-search process compares the held voltage against internal reference levels, step by step, until it finds the digital value that best represents the input voltage.

The resolution of the ADC (N bits) determines how many digital values are available. The input range defines the minimum and maximum voltages the ADC can measure. The size of one least significant bit (LSB) is equal to the input range divided by 2N2^N2N. For accurate measurements, the sampling frequency and sample time must be long enough for the sample-and-hold capacitor to fully charge through the signal's source impedance. If the capacitor does not charge completely, the stored voltage will be incorrect, and the resulting digital value will also be wrong.

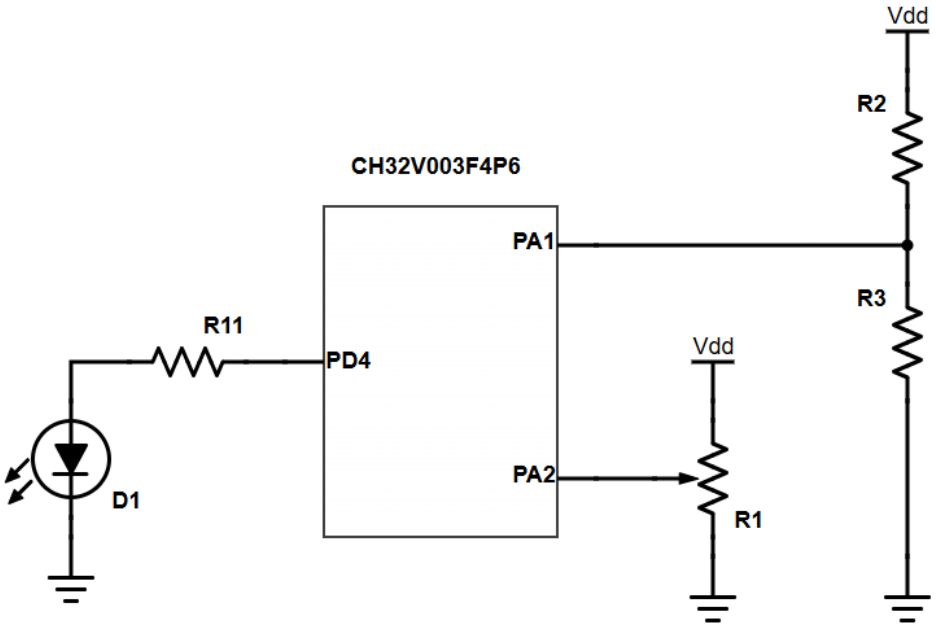
The CH32V003 family implements a 10-bit SAR ADC with eight external channels and two internal sources, designed to sample voltages from ground up to VDDA. The peripheral supports single and continuous conversions, channel scans, externally triggered regular and injected groups, an analog watchdog with programmable thresholds, self-calibration, and a small but useful trigger delay stage. These capabilities are summarized in the reference manual's feature list and block diagram, which also shows the regular data register for standard conversions and a four-deep set of injected data registers, as well as the internal Vref/Vcal paths and the external trigger selectors tied to TIM1/TIM2 lines.

```c
/* ---- ADC ---- */

/* ==== Choose your analog pin/channel here ==== */
/* PC4 is used in this example */
#define ADC_GPIO_PORT GPIOC
#define ADC_GPIO_PIN GPIO_Pin_4
#define ADC_CHANNEL ADC_Channel_2 /* PC4 -> CH2 (typical mapping) */

/* ==== ADC clock: keep <= ~6 MHz (datasheet limit area) ==== */
static void ADC1_Init_Simple(void)
{
 GPIO_InitTypeDef gpio = {0};
 ADC_InitTypeDef adc = {0};

 /* Clocks */
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC | RCC_APB2Periph_ADC1, ENABLE);

 /* Analog pin */
 gpio.GPIO_Mode = GPIO_Mode_AIN;
 gpio.GPIO_Pin = ADC_GPIO_PIN;
 GPIO_Init(ADC_GPIO_PORT, &gpio);

 /* ADCCLK = PCLK2/6 keeps it well within limit at 48 MHz sysclk */
 RCC_ADCCLKConfig(RCC_PCLK2_Div6);

 /* ADC config: single channel, single conversion, right-aligned */
 adc.ADC_Mode = ADC_Mode_Independent;
 adc.ADC_ScanConvMode = DISABLE;
 adc.ADC_ContinuousConvMode = DISABLE;
 adc.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
 adc.ADC_DataAlign = ADC_DataAlign_Right;
 adc.ADC_NbrOfChannel = 1;
 ADC_Init(ADC1, &adc);

 ADC_Cmd(ADC1, ENABLE);

 /* Calibrate (classic STM32-style API carried by WCH's SPL) */
 ADC_ResetCalibration(ADC1);
 while(ADC_GetResetCalibrationStatus(ADC1));
 ADC_StartCalibration(ADC1);
 while(ADC_GetCalibrationStatus(ADC1));
}

/* One blocking conversion on selected channel; 10-bit result (0..1023) */
static uint16_t ADC1_ReadOnce(uint8_t ch)
{
 ADC_RegularChannelConfig(ADC1, ch, 1, ADC_SampleTime_241Cycles);
```

```
 ADC_SoftwareStartConvCmd(ADC1, ENABLE);
 while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
 ADC_ClearFlag(ADC1, ADC_FLAG_EOC);
 return ADC_GetConversionValue(ADC1);
}

/* Public helpers you can call from main */
void ADC_Begin(void) { ADC1_Init_Simple(); }

uint16_t ADC_ReadAveraged(uint8_t ch, uint8_t samples)
{
  uint32_t acc = 0;
  for(uint8_t i=0;i<samples;i++){
  acc += ADC1_ReadOnce(ch);
  }
  return (uint16_t)(acc / samples);
}

/*******************************************************************************
* Function: Main
* Returns: Nothing
* Description: Program entry point
*******************************************************************************/
int main(void)
{
   /* Clock/Delay/UART init (per WCH examples) */
   SystemCoreClockUpdate();
   Delay_Init();
   #if (SDI_PRINT == SDI_PR_OPEN)
    SDI_Printf_Enable();
   #else
    USART_Printf_Init(115200);
   #endif

   printf("SystemClk:%ld\r\n", SystemCoreClock);
   printf("CH32V003 + SSD1306 demo\r\n");

    /* I2C + OLED init */
   I2C1_Init_CH32(100000); /* start at 100 kHz; you can try 400 kHz later */
   Delay_Ms(10);
   OLED_Init();
   Delay_Ms(10);
   OLED_SetOrientation(OLED_ORIENT_180); // rotate 180°
   Delay_Ms(10);
   OLED_Clear();

   /* Demo variables */
```

```
 uint32_t counter = 0;
 uint8_t battery = 100;

 ADC_Begin();

while (1)
{
  /* 16-sample average on PC4 (ADC_CHANNEL) */
  uint16_t raw = ADC_ReadAveraged(ADC_CHANNEL, 16);

  /* 10-bit ADC => 0..1023; estimate mV assuming VDD = 3.30V */
 uint32_t mv = (uint32_t)raw * 3300u / 1023u;

 OLED_Printf_Line(0, "ADC raw: %4u", raw);
 OLED_Printf_Line(1, "ADC mv : %4lu", (unsigned long)mv);

  Delay_Ms(500);
 }
}
```

Lab #30: Comparators



Microcontrollers are not only responsible for digital processing but also for interfacing with the analog world. While the ADC provides a means to digitize analog voltages, sometimes a full conversion is unnecessary. In such cases, comparators serve as fast, low-power, and hardware-efficient decision-making devices. A comparator is essentially a 1-bit ADC: it continuously compares two input voltages and produces a binary digital output that indicates which input is greater.

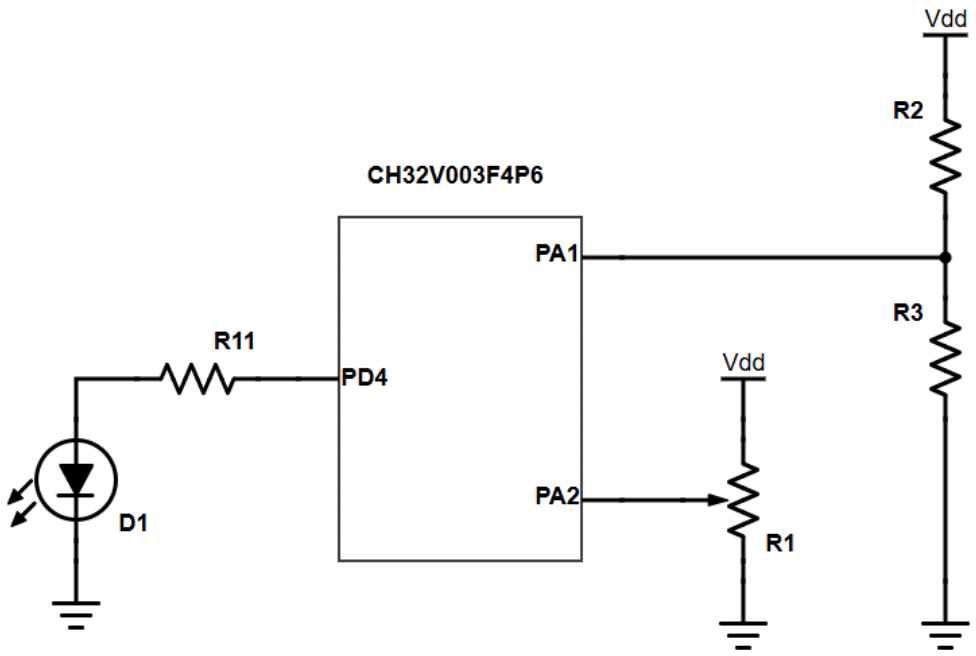
In case you never dealt with the comparator I'll go over it now. The comparator has two input terminals: a non-inverting input (labeled V+) and an inverting input (labeled V–). Its output reflects the relative magnitude of these voltages. If V+ is higher than V–, the comparator drives its output high; if V+ is lower, the output goes low. In the CH32V003, the comparator output is compatible with digital logic, which means it can directly feed into peripheral subsystems such as timers, interrupts, or external pins.

The CH32V003 includes a dedicated comparator peripheral integrated into the analog subsystem. This peripheral allows fast, hardware-based threshold detection without requiring an ADC conversion cycle. The comparator can be configured to compare an external analog signal (from an input pin) against either another pin or an internal reference voltage.

```c
/*********************************************************************
*Includes and defines
*********************************************************************/
#include "debug.h"

// 1) Make sure we are not using HSE so PA1/PA2 can be GPIO/analog
static void use_internal_clock_only(void)
{
  // Enable HSI (internal RC) and wait ready
  RCC->CTLR |= RCC_HSION;
  while(!(RCC->CTLR & RCC_HSIRDY));

  // Switch SYSCLK to HSI
 RCC->CFGR0 &= ~RCC_SWS; // clear status (read-only in many MCUs; fine to ignore)
 RCC->CFGR0 &= ~RCC_SW; // clear switch bits
 RCC->CFGR0 |= RCC_SW_HSI; // select HSI as system clock
 while((RCC->CFGR0 & RCC_SWS) != RCC_SWS_HSI);

 // Disable HSE so the pins aren't reserved by the oscillator block
 RCC->CTLR &= ~RCC_HSEON;
 // (Optional) Disable PLL if it was using HSE
 RCC->CTLR &= ~RCC_PLLON;
}

// 2) Configure PA1/PA2 as analog inputs (high-Z, no digital buffer)
static void config_pa1_pa2_as_analog(void)
{
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
 GPIO_InitTypeDef gi = {0};
 gi.GPIO_Pin = GPIO_Pin_1 | GPIO_Pin_2;
 gi.GPIO_Mode = GPIO_Mode_AIN; // analog mode = no pull, no digital Schmitt trigger
 // Speed is ignored for AIN, but set something valid:
 gi.GPIO_Speed = GPIO_Speed_50MHz;
 GPIO_Init(GPIOA, &gi);
}

// 3) Enable OPA as comparator: PA2 = V+, PA1 = V-
static void opa_enable_pa2_vs_pa1(void)
{
  // Ensure EN, and select PA2 (PSEL=0), PA1 (NSEL=0)
  EXTEN->EXTEN_CTR |= EXTEN_OPA_EN;
  EXTEN->EXTEN_CTR &= ~EXTEN_OPA_PSEL; // 0 -> PA2 as positive
  EXTEN->EXTEN_CTR &= ~EXTEN_OPA_NSEL; // 0 -> PA1 as negative
}

void init_for_comparator_on_pa1_pa2(void)
{
```

```
 use_internal_clock_only(); // stop HSE so PA1/PA2 are free
 config_pa1_pa2_as_analog(); // put pins in analog mode
 opa_enable_pa2_vs_pa1(); // turn on comparator
}

/*******************************************************************************
* Function: Main
*
* Returns: Nothing
*
* Description: Program entry point
*******************************************************************************/
int main(void)
{
 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
  SystemCoreClockUpdate();
  Delay_Init();

 // LED on PD0 as before...
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
 GPIO_InitTypeDef g = {0};
 g.GPIO_Pin = GPIO_Pin_0;
 g.GPIO_Mode = GPIO_Mode_Out_PP;
 g.GPIO_Speed = GPIO_Speed_30MHz;
 GPIO_Init(GPIOD, &g);

  init_for_comparator_on_pa1_pa2();

while(1)
{
 // Easiest: read the OPA output pin PD4 (wire LED/scope to PD4),
 // or mirror it in software if PD4 reads high/low:
 }
}
```

Bonus Lab #31: Direct Memory Access



Direct Memory Access, or DMA, is a feature found in most modern microcontrollers and processors that allows data to be transferred directly between memory and peripherals without needing the central processing unit (CPU) to get involved in every single step. Normally, when a microcontroller wants to move data, for example, reading sensor values from an analog-to-digital converter or sending bytes out through a serial port, the CPU has to actively fetch each piece of data and then store it in memory or send it out to the hardware.

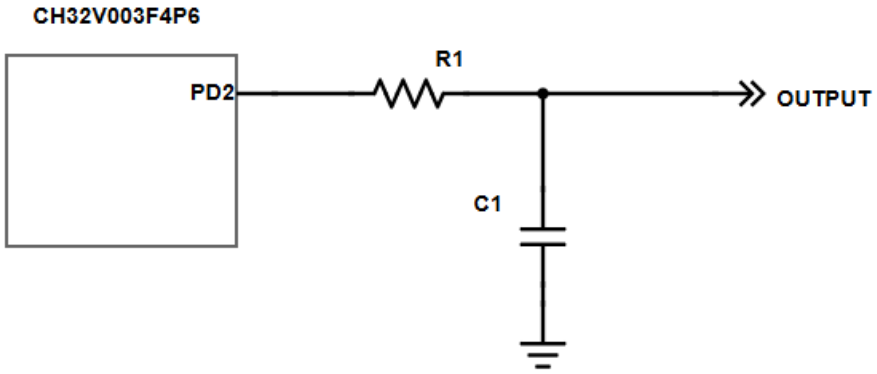
This can be very inefficient because the CPU spends time doing repetitive and low-level tasks instead of focusing on higher-level control or decision-making. With DMA, these transfers are handled automatically by dedicated hardware inside the microcontroller, freeing up the CPU to perform more meaningful work. This makes DMA especially important in applications that require real-time performance or must deal with large amounts of data quickly, such as audio processing, video streaming, or generating continuous waveforms. At its core, DMA works by setting up a small controller inside the microcontroller that can "take over" the system bus, which is the communication pathway between memory and peripherals.

```
/*******************************************************************************
*Includes and defines
*******************************************************************************/
#include "debug.h"

/* Global define */
#define Buf_Size 32

/* Global Variable */
u32 SRC_BUF[Buf_Size] = {0x01020304, 0x05060708, 0x090A0B0C, 0x0D0E0F10,
0x11121314, 0x15161718, 0x191A1B1C, 0x1D1E1F20,
0x21222324, 0x25262728, 0x292A2B2C, 0x2D2E2F30,
0x31323334, 0x35363738, 0x393A3B3C, 0x3D3E3F40,
0x41424344, 0x45464748, 0x494A4B4C, 0x4D4E4F50,
0x51525354, 0x55565758, 0x595A5B5C, 0x5D5E5F60,
0x61626364, 0x65666768, 0x696A6B6C, 0x6D6E6F70,
0x71727374, 0x75767778, 0x797A7B7C, 0x7D7E7F80};

u32 DST_BUF[Buf_Size] = {0};
u8 Flag = 0;
/*******************************************************************
* @fn BufCmp
*
* @brief Compare the buf
*
* @param buf1 - pointer of buf1
* buf2 - pointer of buf1
* buflength - length of buf
*
* @return 1 - Two arrays are identical
* 0 - Two arrays are inconsistent
*/
u8 BufCmp(u32 *buf1, u32 *buf2, u16 buflength)
{
  while(buflength--)
  {
    if(*buf1 != *buf2)
    {
      return 0;
    }
    buf1++;
   buf2++;
  }
  return 1;
}

/*******************************************************************
```

```
* @fn DMA1_CH3_Init
*
* @brief Initializes Channel3 of DMA1 collection.
*
* @return none
*/
void DMA1_CH3_Init(void)
{
 DMA_InitTypeDef DMA_InitStructure = {0};
 RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);

 DMA_StructInit(&DMA_InitStructure);
 DMA_InitStructure.DMA_PeripheralBaseAddr = (u32)(SRC_BUF);
 DMA_InitStructure.DMA_MemoryBaseAddr = (u32)DST_BUF;
 DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
 DMA_InitStructure.DMA_BufferSize = Buf_Size * 4;
 DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
 DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
 DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
 DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
 DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
 DMA_InitStructure.DMA_Priority = DMA_Priority_VeryHigh;
 DMA_InitStructure.DMA_M2M = DMA_M2M_Enable;
 DMA_Init(DMA1_Channel3, &DMA_InitStructure);
 DMA_ClearFlag(DMA1_FLAG_TC3);

 DMA_Cmd(DMA1_Channel3, ENABLE);
}

/*********************************************************************
* @fn main
*
* @brief Main program.
*
* @return none
*/
int main(void)
{
 u8 i = 0;

 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
 SystemCoreClockUpdate();
 Delay_Init();
 #if (SDI_PRINT == SDI_PR_OPEN)
  SDI_Printf_Enable();
 #else
 USART_Printf_Init(115200);
```

```
  #endif
  printf("SystemClk:%d\r\n", SystemCoreClock);
  printf( "ChipID:%08x\r\n", DBGMCU_GetCHIPID() );

  printf("DMA MEM2MEM TEST\r\n");
  DMA1_CH3_Init();

  while(DMA_GetFlagStatus(DMA1_FLAG_TC3) == RESET)
  {
  }

 Flag = BufCmp(SRC_BUF, DST_BUF, Buf_Size);
 if(Flag == 0)
 {
  printf("DMA Transfer Fail\r\n");
 }
 else
 {
    printf("DMA Transfer Success\r\n");
 }

  printf("SRC_BUF:\r\n");
  for(i = 0; i < Buf_Size; i++) {
    printf("0x%08x\r\n", SRC_BUF[i]);
  }

   printf("DST_BUF:\r\n");
   for(i = 0; i < Buf_Size; i++){
     printf("0x%08x\r\n", DST_BUF[i]);
   }
   while(1)
   {
   }
}
```

Bonus Lab #32: Sine Wave



We'll synthesize a sine wave by streaming a lookup table (LUT) of duty values into a PWM channel using DMA. The PWM timer runs at a high carrier frequency (tens of kHz) to ensure the post-filtered output looks analog, while a timer update event (UEV) clocks the DMA so one LUT sample is written to the compare register at a fixed sample rate. The sine's output frequency is simply sample_rate / table_length, so we can tune the pitch without touching DMA by changing the timer's pacing (or the table size). To avoid duty rail hits and dead-time artifacts we pre-scale and center the LUT to use ~10–90% of ARR (leaving headroom for slew, measurement, and gate delays).

Frequency planning starts from the timer base, with a 48 MHz timer clock and prescaler = 0, the PWM carrier is fPWM = 48 MHz / (ARR+ 1). We still want a slower sample clock to step through the LUT, rather than creating a second timer, we reuse the repetition counter (RCR) so that only ever RCR + 1th PWM period produces a UEV → DMA write. This clearly decimates the effective update rate without sacrificing the carrier frequency. With ARR = 832 (=57.6 PWM) and RCR = 8, the UEV rate is about 57.6 kHz / 9 = 6.4 kHz. With a 32-sample table the sine fundamental lands at about 6.4 kHz / 32 = 200 Hz which is an easy to measure audio-ish tone.

```c
/*********************************************************************************
*Includes and defines
*********************************************************************************/
#include "debug.h"

/* ===== Config ===== */
#define SINE_SAMPLES 32
#define ARR_VALUE 832 // ~57.6 kHz PWM carrier @48MHz (PSC=0)
#define PSC_VALUE 0
#define RCR_VALUE 8 // Update every (RCR+1)=9 PWM periods -> ~6.4 kHz sample => ~200 Hz sine

/* ===== Base 12-bit sine (0..4095), 32 samples ===== */
static const uint16_t SINE_BASE12[SINE_SAMPLES] = {
  2048,2447,2831,3185,3495,3750,3939,4056,
  4095,4056,3939,3750,3495,3185,2831,2447,
  2048,1649,1265, 911, 601, 346, 157, 40,
  0, 40, 157, 346, 601, 911,1265,1649
};

/* Scaled/centered LUT (10..90% of ARR, avoids rail hits) */
static uint16_t SINE_WAVE[SINE_SAMPLES];

/* Build centered headroom LUT from base (integer, no libm)   mid = ARR/2, amp = 0.8*(ARR/2) -> range ≈ 10%..90% of ARR */
static void gen_sine_lut_scaled(uint16_t arr)
{
  uint32_t mid = arr / 2u;
  uint32_t amp = (arr / 2u) * 80u / 100u; // 0.8 * (arr/2)
  uint32_t two_amp = amp * 2u;

  for (int i = 0; i < SINE_SAMPLES; i++) {
    uint32_t x = SINE_BASE12[i]; // 0..4095
    uint32_t offset = (two_amp * x + 2047u) / 4095u; // ≈ 2*amp*(x/4095) with rounding
     int32_t v = (int32_t)(mid - amp) + (int32_t)offset;
     if (v < 0) v = 0;
     if (v > (int32_t)arr) v = (int32_t)arr;
     SINE_WAVE[i] = (uint16_t)v;
   }
}

/* TIM1 CH1 compare register (WCH naming) */
#define TIM1_CH1CVR_ADDR ((uint32_t)&TIM1->CH1CVR)

/* ===== PD2 as TIM1_CH1 ===== */
static void GPIO_PD2_TIM1CH1_Init(void)
{
  GPIO_InitTypeDef io = {0};
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
```

```
  io.GPIO_Pin = GPIO_Pin_2;
  io.GPIO_Mode = GPIO_Mode_AF_PP;
  io.GPIO_Speed = GPIO_Speed_30MHz;
  GPIO_Init(GPIOD, &io);
}

/* ===== TIM1 base + PWM1 on CH1 (with RCR decimation) ===== */
static void TIM1_PWM_Init(uint16_t arr, uint16_t psc, uint8_t rcr)
{
 TIM_TimeBaseInitTypeDef tb = {0};
 TIM_OCInitTypeDef oc = {0};

 RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE);

 tb.TIM_Period = arr;
 tb.TIM_Prescaler = psc;
 tb.TIM_ClockDivision = TIM_CKD_DIV1;
 tb.TIM_CounterMode = TIM_CounterMode_Up;
 tb.TIM_RepetitionCounter = rcr; // decimate UEV -> DMA pacing
 TIM_TimeBaseInit(TIM1, &tb);

 oc.TIM_OCMode = TIM_OCMode_PWM1; // duty = CH1CVR / ARR
 oc.TIM_OutputState = TIM_OutputState_Enable;
 oc.TIM_Pulse = 0;
 oc.TIM_OCPolarity = TIM_OCPolarity_High;
 TIM_OC1Init(TIM1, &oc);

 TIM_OC1PreloadConfig(TIM1, TIM_OCPreload_Enable); // latch at update
 TIM_ARRPreloadConfig(TIM1, ENABLE);
}

/* ===== DMA: mem -> TIM1->CH1CVR, circular ===== */
static void TIM1_DMA_Sine_Init(void)
{
 DMA_InitTypeDef d = {0};
 RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
 DMA_DeInit(DMA1_Channel5); // TIM1_UP -> CH5 on CH32V003

 d.DMA_PeripheralBaseAddr = TIM1_CH1CVR_ADDR; // dest
 d.DMA_MemoryBaseAddr = (uint32_t)SINE_WAVE; // src (scaled/centered)
 d.DMA_DIR = DMA_DIR_PeripheralDST;
 d.DMA_BufferSize = SINE_SAMPLES;
 d.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
 d.DMA_MemoryInc = DMA_MemoryInc_Enable;
 d.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
 d.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
 d.DMA_Mode = DMA_Mode_Circular;
```

```
 d.DMA_Priority = DMA_Priority_VeryHigh;
 d.DMA_M2M = DMA_M2M_Disable;
 DMA_Init(DMA1_Channel5, &d);
 DMA_Cmd(DMA1_Channel5, ENABLE);
 TIM_DMACmd(TIM1, TIM_DMA_Update, ENABLE); // one transfer per UEV
}

int main(void)
{
 SystemCoreClockUpdate(); // 48 MHz expected
 Delay_Init();
 #if (SDI_PRINT == SDI_PR_OPEN)
  SDI_Printf_Enable();
 #else
   USART_Printf_Init(115200);
 #endif
  printf("\r\nCH32V003 TIM1+DMA sine on PD2 (centered 10..90%%)\r\n");
  printf("SystemClk:%ld\r\n", SystemCoreClock);

  gen_sine_lut_scaled(ARR_VALUE); // build headroom LUT

  GPIO_PD2_TIM1CH1_Init();
  TIM1_PWM_Init(ARR_VALUE, PSC_VALUE, RCR_VALUE);
  TIM1_DMA_Sine_Init();

  TIM_Cmd(TIM1, ENABLE);
  TIM_CtrlPWMOutputs(TIM1, ENABLE); // MOE for advanced timer

  while (1) { /* DMA + TIM do the work */ }
}
```