# Unlocking the Potential of Your Microcontroller

Ethan Wu

Storming Robots, Branchburg NJ, USA

**Abstract.** Many useful hardware features of commonly-used advanced microcontrollers are often not utilized to their fullest potential, requiring workarounds such as multiple microcontrollers. By allowing microcontroller hardware to handle interfacing with most robot components, including sampling sensors, driving motors, and sending or receiving other data, the microcontroller's core is free to focus on other tasks such as compute-intensive processing and algorithms. Thus a robot can be built with the same functionality while utilizing fewer independent microcontrollers as well as having the potential for greater performance.

## 1 Introduction

For any robot, some form of microcontroller is responsible for controlling its actions. With the numerous sensors, motors, and actuators found on robots, the microcontrollers have a lot of data that they need to quickly and accurately acquire and process. However, the most commonly used microcontroller, the ATmega328 (found in most Arduinos) typically struggles with handling the necessary sensors while still allotting enough time to meet processing deadlines. Thus more powerful microcontrollers are used, changing from 8-bit to 32-bit systems that offer much more raw compute power in the form of clock speed as well as more functionality and on-board peripherals. But the Arduino ecosystem remains the most popular way to program these devices simply because developers are accustomed to its libraries, which were initially designed for more limited systems and ease of entry. Therefore many of the hardware features of these more advanced microcontrollers are left unused; often kludges—such as adding more microcontrollers—are employed. This paper will examine how a single microcontroller can be sufficient if the potential of its hardware is fully exploited.

## 2 Basic hardware

### 2.1 GPIOs

The most basic peripheral of any microcontroller is the GPIOs, or General Purpose Input/Outputs. These are the digital pins on a microcontroller, which can of course read a digital line (measuring voltage) or write to a digital line (generating voltage). They can be directly controlled by the core, but can also be

assigned to the inputs and outputs of other peripherals on the microcontroller. For example, on the STM32 series of microcontrollers, all non-special pins (i.e. everything except power and clock source) are GPIOs that can be assigned to be used by other peripherals available on that pin, such as ADCs, I2C, and even the debug port. Thus, one physical pin on the microcontroller package can serve many different roles, making pins with many peripherals attached valuable for future expansion and alternative uses or applications.

### 2.2 Interrupts

Interrupts are one of the most basic and important concepts in an embedded microcontroller; they have many applications, discussed throughout this paper. There are many forms of interrupts, which many different components can trigger, but they all serve to "interrupt" other hardware in the microcontroller, essentially informing it that some event happened. Arduino users are likely familiar with interrupt pins—interrupts triggered by a changing digital state on a GPIO pin. These external interrupts can trigger on a variety of conditions, including at minimum rising and falling edge triggers [3]. They can trigger an interrupt service routine in the microcontroller's core, which is what is typically seen with Arduinos, but can also trigger other hardware and peripherals inside of the microcontroller.

Besides physical pins and external events, other internal peripherals inside of the microcontroller can also trigger interrupts, typically signaling an event that occurred in the peripheral. This enables complex hardware-handled behavior by allowing various peripherals to work together asynchronously, responding to tasks as they need to be done.

## 3 Timers

### 3.1 Functional description

One of the most basic and versatile features on a microcontroller is its timers. They perform a very simple function—count. However, the various triggers, counting modes, and output options are what makes them so powerful. Timers typically contain one or more capture/compare registers (CCRs), which, as their name implies, can either capture from the timer's counter or compare with it. In input capture mode, the register can be set to save the current value of the counter when an edge is detected on a connected GPIO (configurable as either rising or falling). This is particularly useful for capturing precise timing information without any intervention from the core. The CCR can effectively measure a time and "latch" the result, so the core can read the CCR when it needs the value. The CCR can also act in output compare mode, where a GPIO pin's state is set depending on the value of the timer's counter and the CCR itself; if the counter is either greater than, less than, or equal to the value of the CCR the output would be set either high or low, all according to the configuration. Finally, the timer's counter itself can also be controlled; one timer can be slaved to

another, only start on an interrupt, only count through once, only count when a trigger line is high or low, or get reset by an interrupt. [2]

### 3.2 Example uses

Using the functions described previously, one can perform many tasks with only a timer and no involement from the core.

**PWM output** PWM output is one such task commonly utilizing timers, and is essential to many basic robot functions such as motor control. Arduino's hardware PWM out uses a timer, with the CCR in output compare mode to bring the PWM line high on a certain portion of the counter cycle. Because the timers can use very high clock speeds for their counter source, this method can generate extremely high frequency PWM; for example, a STM32F405's timer can run at up to 42MHz [2].

**PWM input** A timer can also be used to capture information about an incoming PWM waveform, providing period and duty cycle without using ADCs and filter capacitors or any involvement from the core. Two CCRs, one for rising and one for falling edge, combined with a edge-triggered counter reset means one CCR captures the on-time, while the other captures the period of the PWM signal. Because the timer's counter can be "wired" directly to the microcontroller's main clock (with dividers and prescalers), timing measurements are very accurate (if the oscillator is accurate). To read the captured PWM data, the core simply needs to read the CCRs over the memory bus, with no further action required after setup. For example, this capture setup can be used with Melexis non-contact temperature sensors often used for RCJ Rescue Maze; these sensors can be configured to output temperature as PWM. This output can then be connected with a timer to have acquiring temperature readings done entirely by microcontroller hardware and independently for multiple sensors, without a need for (typically sequential) polling over an I2C or SPI bus.

**Encoders** Many microcontrollers also contain timers supporting an encoder mode, which utilizes a timer's counter and counter control triggers to automatically count up and down using interrupts on a standard quadrature encoder signal. This significantly improves upon simply utilizing external interrupts because it again needs no interaction with the core to count encoder steps. The core simply needs to read the counter register when an encoder value is desired (and potentially reset the counter), and because the counting is handled by hardware, it is almost impossible to skip encoder steps. This contrasts with an interrupt-based scheme typically seen on Arduinos, where a high rate of encoder steps or simply the core being busy with other interrupts can cause encoder steps to be lost [1].

**Pulse capture** Timers' slave functionality can also be used to interface with sensors. For instance, timers can be used to capture the length of a standard ultrasonic sensor's echo pulse—where the sensor raises a line high for the amount of time it took an ultrasonic pulse to transmit and return. A timer could be run in gated mode, meaning the counter only increments when a trigger is high—thus counting the time the ultrasonic pulse was high. This would have to be paired with the core triggering the ping and later resetting the counter, but would still benefit from the timing accuracy of the timer. Alternatively, the timer could also use a CCR to capture the falling edge and reset the counter on the rising edge, which does not need resetting from the microcontroller. Completely hardware-controlled ultrasonic sensor ranging can be done by simply using another timer to trigger the pulses.

## 4 Peripheral features

### 4.1 ADC

Almost all microcontroller contain at least one analog-to-digital converter (ADC). As their name suggests, they convert analog voltages into a digital number for the core to process. More advanced microcontrollers also have additional features on ADCs such as analog watchdogs, which will generate an interrupt when a certain channel's value goes outside of a predefined range. ADCs can also include various sampling modes and sampling controls. The most basic of these is the sample rate; higher rates can be achieved by using fewer bits of precision. Many ADCs can also set up sampling sequences, allowing multiple channels to be sampled with their own sampling times, and the sampling order to be customized. For example, if a channel needs to be sampled more frequently than the others, it could occupy more sampling slots in the ADC's scan sequence (e.g. 1, 2, 1, 3, 1, 4); if a channel needed additional accuracy, it could have a longer sampling time.

### 4.2 Serial protocols

Microcontrollers contain various hardware peripherals for common serial protocols, such as $I^2C$, SPI, USART (which supports both UART or standard "serial" and more obscure synchronous protocols), and CAN. These peripherals are likely familiar to Arduino users and handle transmitting data over the protocol. The peripheral typically performs this one word (usually byte) at a time, while handling protocol-level logic such as flow-control in UART and clock lines for $I^2C$ and SPI. They will fire interrupts for when the transmit register is done transmitting for the next word to be loaded into the register. Similarly, incoming data often also triggers an interrupt for the data to be read out of the receive register before the next word is received.

## 5 DMA

Many microcontrollers also possess Direct Memory Access peripherals, which copy memory from one region to another. This is typically used to move data between a peripheral and main memory, and is particularly useful for "queuing" up data to a peripheral that can only accept one word at a time. Each DMA consists of streams, each of which will transfer between specific regions when an interrupt is fired. Most peripherals are able to trigger DMA requests when appropriate; for example, an ADC will fire the interrupt when a conversion is finished, and a transmitting I2C device will fire the interrupt once the current byte has been processed.

The DMA channels can be configured to automate many "long" tasks that would typically involve the core repeatedly writing or reading data. An ADC can read all of its input channels into main memory without core involvement by configuring DMA to cycle through memory addresses; this spreads out the one ADC output register into memory where each ADC channel has its own address, and is updated independently. Thus any necessary ADC values can be updated in the background. Communications over serial ports (UART, I2C, SPI, etc.) can also be streamlined with DMA, because the DMA can take care of sequentially "feeding" words to the peripheral from a memory buffer. The DMA request is fired after each word is transmitted, and the DMA then copies the next word from memory to the transmit register. The core simply needs to start this request, and does not need further involvement in transmission, making this operation completely asynchronous and handled by hardware.

## 6 Conclusion

By utilizing more of the hardware capabilities of a microcontroller and choosing sensors to suit these capabilities, one can greatly streamline data acquisition for a robot. Most robots require some form of distance sensors; the common Sharp IR range finders come in an analog output mode, which can be connected with spare ADC ports to capture all ranging data with microcontroller hardware. For example, on a robot for RCJ Rescue Maze, 6 distance sensors (using analog), 2 temperature sensors (PWM in), 3 motors (servos included—PWM out), 2 quadrature encoders, an ultrasonic sensor (pulse capture), and other analog sensors can all be controlled or have data captured without intervention from the core (see Fig. 1—only $I^2C$ devices require action from the core). This frees the core to perform many other processing-intensive tasks—such as complex IMU sensor fusion—without missing data or blocking to read data. This intelligent design allowed for an extremely complex robot utilizing many data sources and compute-intensive algorithms to run all of its real-time processing off a single microcontroller.

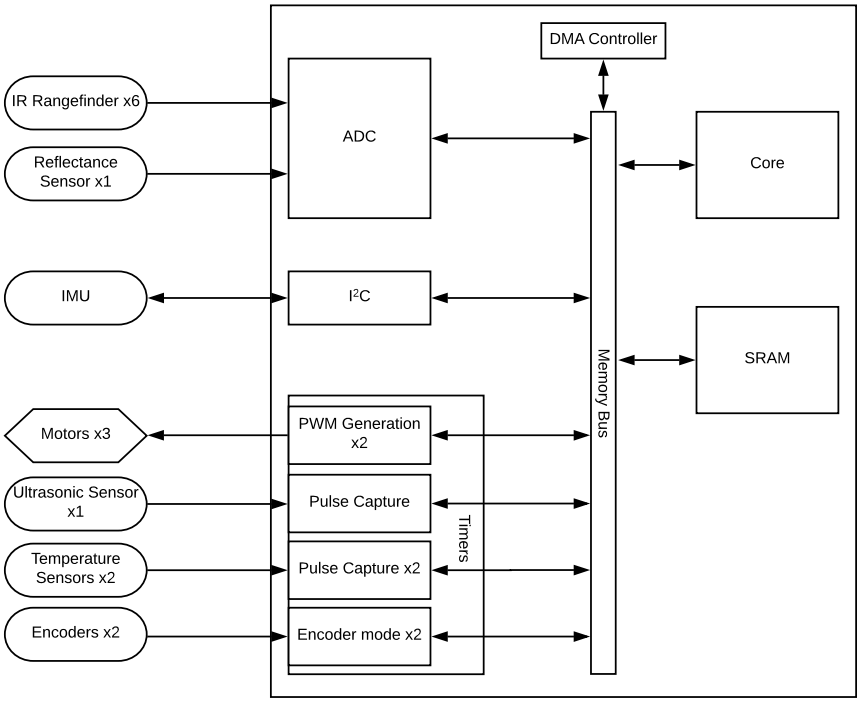


**Fig. 1.** Example of a RCJ Rescue Maze robot architecture utilizing hardware peripherals of a microcontroller

## References

1. Anderson, R., Cervo, D.: Pro Arduino. Apress, Berkeley, CA (2013)
2. STMicroelectronics: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs, 16 edn. (April 2018)
3. White, E.: Making embedded systems. O'Reilly, Beijing (2012)