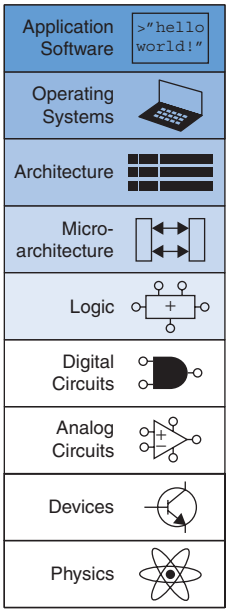
e9 Embedded I/O Systems

9.1 INTRODUCTION

Input/Output (I/O) systems are used to connect a computer with external devices called *peripherals*. In a personal computer, the devices typically include keyboards, monitors, printers, and wireless networks. In embedded systems, devices could include a toaster's heating element, a doll's speech synthesizer, an engine's fuel injector, a satellite's solar panel positioning motors, and so forth. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

This chapter provides concrete examples of I/O devices. Section 9.2 shows the basic principles of interfacing an I/O device to a processor and accessing it from a program. Section 9.3 examines I/O in the context of embedded systems. It shows how to use SparkFun's RED-V RedBoard, which has a RISC-V microcontroller, to access on-board peripherals including general-purpose, serial, and analog I/O as well as timers and pulse-width modulation (PWM). Section 9.4 gives examples of interfacing with other common devices, such as character LCDs, VGA monitors, Bluetooth radios, and motors.



9.2 MEMORY-MAPPED I/O

Recall from Section 6.5.1 that a portion of the address space is dedicated to I/O devices rather than memory. For example, suppose that physical addresses in the range 0x20000000 to 0x20FFFFFF are used for I/O. Each I/O device is assigned one or more memory addresses in this range. A store to the specified address sends data to the device. A load receives data from the device. This method of communicating with I/O devices is called *memory-mapped* I/O.

In a system with memory-mapped I/O, a load or store may access either memory or an I/O device. Figure e9.1 shows the hardware needed to support two memory-mapped I/O devices. An address decoder determines which device communicates with the processor. It uses the *Address* and *MemWrite* signals to generate control signals for the rest of

Digital Design and Computer Architecture, RISC-V Edition. DOI: 10.1016/B978-0-12-820064-3.00019-2

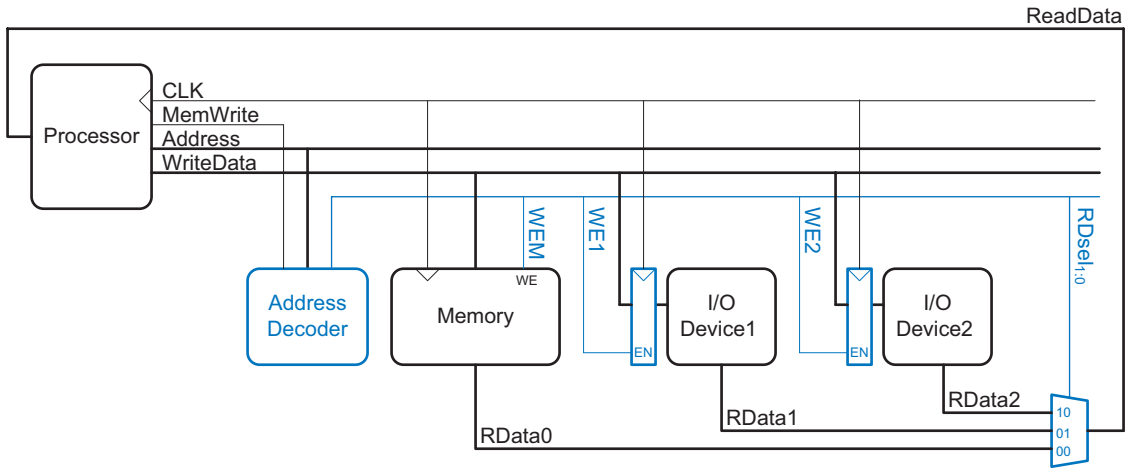


Figure e9.1 **Support hardware for memory-mapped I/O**

the hardware. The ReadData multiplexer selects between memory and the various I/O devices. Write-enabled registers hold the values written to the I/O devices.

Embedded processors are so named because they are typically embedded within a larger system (such as a toy or an automobile) and have a limited user interface. In contrast, processors found in PCs have interfaces such as keyboards and screens that make them accessible to program or run applications. But all types of processors are essentially the same—they all execute instructions. Only the interfaces and peripheral devices used by embedded and traditional processors differ.

Example e9.1 COMMUNICATING WITH I/O DEVICES

Suppose that I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the RISC-V assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution The following RISC-V assembly code writes the value 7 to I/O Device 1. The `.equ` assembler directive replaces the named symbol with the given value. So, the `li s1, ioadr` instruction becomes `li s1, 0x20001000`.

```
.equ ioadr   0x20001000

     li s0, 7
     li s1, ioadr
     sw s0, 0(s1)
```

The address decoder detects address 0x20001000 and *MemWrite* = 1, so it asserts *WE1*, the write enable for Device 1's register. At the next clock edge, the value on the *WriteData* bus, 7, is written into the register, whose output connects to the input pins of I/O Device 1.

To read from I/O Device 1, the processor executes the following RISC-V assembly code.

```
lw s0, 0(s1)
```

The address decoder detects the address 0x20001000, so it sets $RDsel_{1:0}$ to 01. The multiplexer thus selects *RData1*, the read data from Device 1, and connects it to the *ReadData* bus, the value of which is then loaded into `s0` in the processor.

The addresses associated with I/O devices are often called *I/O registers* because they may correspond with physical registers in the I/O device like those shown in Figure e9.1.

Software that communicates with an I/O device is called a *device driver*. You have probably downloaded or installed device drivers for your printer or other I/O device. Writing a device driver requires detailed knowledge about the I/O device hardware, including the addresses and behavior of the memory-mapped I/O registers. Other programs call functions in the device driver to access the device without having to understand the low-level device hardware.

According to IC Insights, approximately 24 billion microcontrollers were sold in 2020, and the market is forecast to grow at 10% per year through 2029. The average price of a standalone microcontroller is about 60 cents, and an 8-bit microcontroller can be integrated on a system-on-chip (SoC) for less than a tenth of a penny. Microcontrollers have become ubiquitous and nearly invisible, with an estimated 100 or more microcontrollers in an average new car in 2021.

Automobiles are the largest and fastest-growing market for microcontrollers, followed by consumer electronics, industrial systems, medical devices, and military applications. 16-bit microcontrollers account for the most revenue in 2020, but 32-bit microcontrollers are increasing in market share because of their greater capabilities.

Leading microcontroller manufacturers are Infineon, Microchip, NXP, Renesas, STMicroelectronics, and Texas Instruments. Leading architectures include the 8051, AVR, PIC, and ARM. ARM holds a near-monopoly as the application processor for 90% of mobile devices. However, RISC-V is gaining great interest as a new and open-source architecture.

9.3 EMBEDDED I/O SYSTEMS

Embedded systems use a processor to control interactions with the physical environment. They are typically built around microcontroller units (MCUs) which combine a microprocessor with a set of easy-to-use peripherals such as general-purpose digital and analog I/O pins, serial ports, timers, etc. Microcontrollers are generally inexpensive and are designed to minimize system cost and size by integrating most of the necessary components onto a single chip. Most are smaller and lighter than a dime, consume milliwatts of power, and range in cost from a few dimes up to several dollars. Microcontrollers are classified by the size of data that they operate on. 8-bit microcontrollers are the smallest and least expensive, while 32-bit microcontrollers provide more memory and higher performance.

9.3.1 RED-V Board

For the sake of concreteness, this section will illustrate embedded system I/O in the context of a real system. Specifically, we will focus on the FE310-G002 system-on-chip (SoC) from SiFive, which contains a 320 MHz 32-bit RISC-V processor that implements the RV32IMAC architecture—that is, the base 32-bit integer instruction set (RV32I) plus the multiply/divide (M), atomic memory accesses (A), and compressed 16-bit instructions (C) extensions. This MCU is available on the HiFive development board from SiFive as well as on a set of third-party development boards such as the RED-V series from SparkFun (available in both Arduino and Thing Plus footprints). The I/O interfaces described in each subsection will be followed by specific examples that run on the FE310. All of the examples have been tested on SparkFun's RED-V

RedBoard and could be readily run on the HiFive development board or adapted to the RED-V Thing Plus Board.

The RED-V RedBoard is called simply *the RED-V* throughout this chapter.

Figure e9.2(a) shows SparkFun's RED-V RedBoard, which is available for less than $40 and is 2.7" × 2.1". The figure also shows each pin's signal names, which we describe throughout this section. The development board can be powered from a 5 V USB power supply or from a 7 to 15 V DC source via the barrel jack. The FE310-G002 on board is powered by 3.3 V and 1.8 V on-board regulators. The FE310-G002 has a 16-KiB L1 Instruction Cache and a 16-KiB Data SRAM Scratchpad. The SparkFun development board also has 32 MiB of off-chip flash storage accessible via a serial peripheral interface (SPI) that can be used to store programs and data.

This book's companion materials (see the Preface) include laboratory exercises that use the RED-V board.

Figure e9.2(b) shows the RED-V Thing Plus, which has capabilities similar to the RED-V RedBoard but in a smaller form factor (2.3" × 0.9") that fits on a breadboard for easy interfacing. The I/O pins are numbered differently than on the RedBoard and are difficult to read on the silk screen, but they are labeled in Figure e9.2(b).

The RED-V RedBoard form factor is designed around an Arduino R3 footprint in an effort to preserve as much compatibility as possible with the many Arduino shields available in this footprint. All 19 configurable I/O signals are accessible via header pins and operate at 3.3 V. The header also provides 3.3 V, 5 V, and ground to conveniently power small devices attached to the RedBoard, but the maximum total current is 50 mA from the 3.3 V supply and ~300 mA from the 5 V supply.

Caution: Connecting 5 V to one of the 3.3 V I/Os may damage the I/O and possibly the entire FE310. If you probe the I/O pins with a voltmeter, beware that you do not accidentally make contact between VUSB or VBAT and a nearby pin!

Maintaining compatibility with the Arduino R3 footprint results in multiple names for each pin: the silkscreen (text printed on the board) lists the standard Arduino pin numbers, but the RED-V pinout documented in Figure e9.2 shows both the Arduino pin numbers and the corresponding FE310 GPIO (general-purpose I/O) pin numbers. For

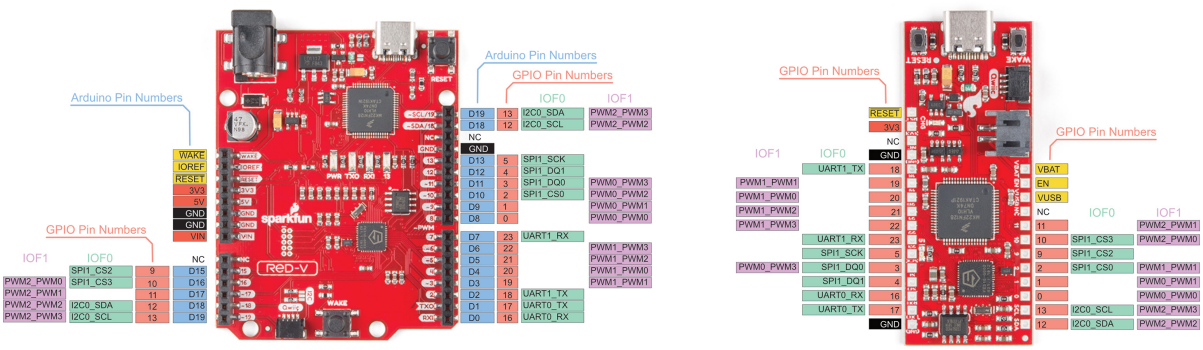


Figure e9.2 (a) RED-V RedBoard; (b) RED-V Thing Plus Board
(Photos courtesy of SparkFun used under CC BY 2.0)

example, as shown in Figure e9.2, GPIO5 (FE310 pin 5) corresponds to D13 (Arduino pin 13). The RED-V Thing Plus board lacks the Arduino-compatible pin naming but, thus, also avoids having multiple pin numbers for a single pin. Also notice in Figure e9.2 that some GPIO pins have multiple purposes. For example, GPIO18 (D2) can also act as the transmit line for UART 1 (UART1_TX), as will be described later.

On both boards, GPIO5 is connected to a blue LED. This pin is labeled 13 (i.e., D13) on the RedBoard and 5 on the Thing Plus board.

This section begins by describing the FE310-G002 SoC and describing a general device driver for memory-mapped I/O. The remainder of this section illustrates how embedded systems perform general-purpose digital, analog, and serial I/O.

SiFive was founded in 2015 by three researchers from the University of California, Berkeley: Krste Asanović, Yunsup Lee, and Andrew Waterman. SiFive's vision is to make custom silicon development faster and more affordable than ever before. Focused around the open RISC-V instruction set architecture (ISA), they have developed a platform that enables system-level design of custom chips. More information, including the FE310-G002 datasheet, can be found at sifive.com.

The RISC-V microcontrollers from SiFive continue to advance. By the time you read this, a newer model might be available with a more advanced processor and a different set of embedded I/O. Nevertheless, the same principles discussed here apply to that microcontroller as well as other microcontrollers. You can expect to find the same types of I/O and peripherals. You will need to consult the datasheet to look up the mapping between the peripheral, the pin on the chip, and the pin on the board, as well as the memory-mapped I/O addresses (registers) associated with each peripheral. But, as described here, you will still write to configuration registers to initialize the peripheral and read and write data registers to communicate with the peripheral.

9.3.2 FE310-G002 System-on-Chip

The FE310-G002 SoC is a powerful yet inexpensive microcontroller chip designed by SiFive. It includes a RISC-V microprocessor with a 5-stage pipeline similar to the one described in Chapter 7 and many I/O peripherals. The FE310 is packaged in a 48-lead, quad flat no-leads package. SiFive publishes a datasheet that describes many features and I/O registers; this chapter discusses only a subset of those features.

Table e9.1 shows the FE310 memory map. Upon start-up, the processor begins executing code from external flash memory at address 0x20000000. The memory map has room for up to 512 MiB of external flash, although current RED-V boards have much less: the RED-V RedBoard has 32 MiB of external flash, and the RED-V Thing Plus has 4 MiB. The chip also has 16 KiB of RAM, called a *data tightly integrated memory* (DTIM), at address 0x80000000. This RAM has a 2-cycle load latency and is used to hold variables. Various peripherals are memory-mapped between addresses 0x02000000 and 0x1FFFFFFF and will be described in detail in later sections. These peripherals include general-purpose I/O, three pulse-width modulation (PWM) blocks for generating output waveforms, and many serial ports to connect to external devices, including three serial peripheral interfaces (SPIs), two universal asynchronous receiver/transmitters (UARTs), and one inter-integrated circuit (I^2C) interface.

Figure e9.3 shows a simplified schematic of the RED-V RedBoard. The board receives 5V power from a USB power supply and regulators produce 3.3V and 1.8V for I/O, powering the low-power always-on core and miscellaneous functions.

9.3.3 General-Purpose Digital I/O

General-purpose I/O (GPIO) pins are used to read or write digital signals. At a minimum, GPIO pins require memory-mapped I/O registers

Table e9.1 FE310 Memory Map

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000	0x0000_1FFF	R XC	Mode Select	On-Chip Non Volatile Memory
0x0000_2000	0x0000_2FFF		Reserved	
0x0000_3000	0x0000_3FFF	RWX A	Error Device	
0x0000_4000	0x0000_FFFF		Reserved	
0x0001_0000	0x0001_1FFF	R XC	Mask ROM (8 KiB)	
0x0001_2000	0x0001_FFFF		Reserved	
0x0002_0000	0x0002_1FFF	R XC	OTP Memory Region	
0x0002_2000	0x001F_FFFF		Reserved	
0x0200_0000	0x0200_FFFF	RW A	CLINT	On-Chip Peripherals
0x0201_0000	0x07FF_FFFF		Reserved	
0x0800_0000	0x0800_1FFF	RWX A	E31 ITIM (8 KiB)	
0x0800_2000	0x0BFF_FFFF		Reserved	
0x0C00_0000	0x0FFF_FFFF	RW A	PLIC	
0x1000_0000	0x1000_0FFF	RW A	AON	
0x1000_1000	0x1000_7FFF		Reserved	
0x1000_8000	0x1000_8FFF	RW A	PRCI	
0x1000_9000	0x1000_FFFF		Reserved	
0x1001_0000	0x1001_0FFF	RW A	OTP Control	
0x1001_1000	0x1001_1FFF		Reserved	
0x1001_2000	0x1001_2FFF	RW A	GPIO	
0x1001_3000	0x1001_3FFF	RW A	UART 0	
0x1001_4000	0x1001_4FFF	RW A	QSPI 0	
0x1001_5000	0x1001_5FFF	RW A	PWM 0	
0x1001_6000	0x1001_6FFF	RW A	I2C 0	
0x1001_7000	0x1002_2FFF		Reserved	
0x1002_3000	0x1002_3FFF	RW A	UART 1	
0x1002_4000	0x1002_4FFF	RW A	SPI 1	
0x1002_5000	0x1002_5FFF	RW A	PWM 1	
0x1002_6000	0x1003_3FFF		Reserved	
0x1003_4000	0x1003_4FFF	RW A	SPI 2	
0x1003_5000	0x1003_5FFF	RW A	PWM 2	
0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

Memory attributes: R: Read, W: Write, X: Execute, C: Cacheable, A: Atomics.
Reprinted with permission from Table 4 of the *FE310-G0002 Manual,* © 2019 SiFive, Inc.

to read input pin values, write output pin values, and set the direction of the pin. In many embedded systems, the GPIO pins can be shared with one or more special-purpose peripherals, so additional configuration registers are necessary to determine whether the pin is general- or special-purpose. Furthermore, the processor may generate interrupts when an event such as a rising or falling edge occurs on an input pin, and configuration registers may be used to specify the conditions for

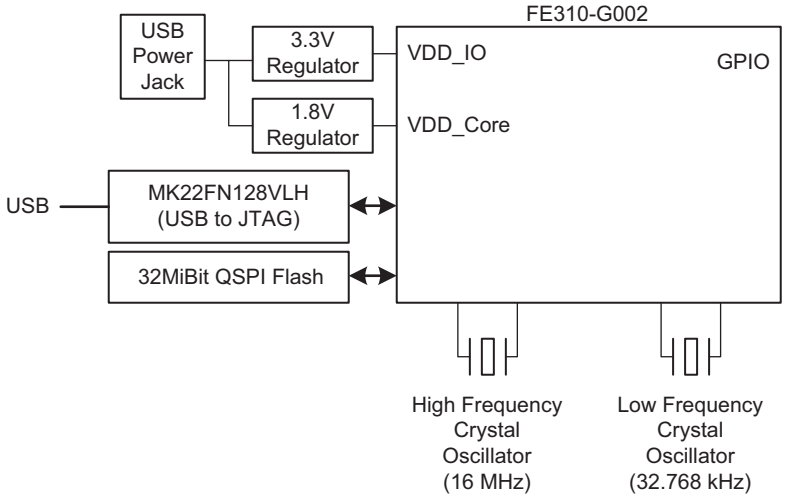


Figure e9.3 RED-V board schematic

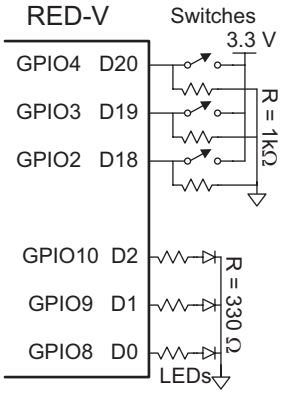


Figure e9.4 GPIO example

an interrupt. Recall that the FE310 has 19 GPIO pins. This section will start with a basic example of controlling these pins and then will look at some of the special purposes for these pins.

Figure e9.4 shows three light-emitting diodes (LEDs) and three switches connected to six GPIO pins. The LEDs are wired to glow when driven to 1 and to turn off when driven to 0. The current-limiting (typically around 300 Ω) resistors are placed in series with the LEDs to set the brightness and to avoid overloading the current capability of the GPIO. The switches are wired to produce a 1 when closed and a 0 when open. As shown, the 1 kΩ pull-down resistors pull the pins down to 0 when the switches are open. Figure e9.4 indicates the (Arduino) pin numbers that are labeled on the board as well as the GPIO pin numbers.

Table e9.2 lists the GPIO registers and their address offsets relative to the GPIO base address, 0x10012000, as shown in Table 51 of the *FE310-G002 Manual*. Let's first focus on the top four registers (i.e., memory-mapped I/O addresses). Each GPIO pin is mapped to one bit of the registers. Reading from the `input_val` (input value) register reads the values of the GPIO pins, and writing to the `output_val` (output value) register writes to the GPIO pins. Before reading or writing to the pins, the input and output enable registers (`input_en` and `output_en`) must be set to configure the pins as inputs or outputs and the hardware-driven function enable register (`iof_en`) must be cleared to configure the pins as GPIO controlled.

GPIO Memory-Mapped I/O

We illustrate how to use the GPIO pins by writing a program that reads the state of a switch and controls an LED using the GPIOs. The

Table e9.2 GPIO register offsets

Offset	Name	Description
0x00	input_val	Pin value
0x04	input_en	Pin input enable*
0x08	output_en	Pin output enable*
0x0C	output_val	Output value
0x10	pue	Internal pull-up enable*
0x14	ds	Pin drive strength
0x18	rise_ie	Rise interrupt enable
0x1C	rise_ip	Rise interrupt pending
0x20	fall_ie	Fall interrupt enable
0x24	fall_ip	Fall interrupt pending
0x28	high_ie	High interrupt enable
0x2C	high_ip	High interrupt pending
0x30	low_ie	Low interrupt enable
0x34	low_ip	Low interrupt pending
0x38	iof_en	HW-driven functions enable
0x3C	iof_sel	HW-driven functions selection
0x40	out_xor	Output XOR (invert)

Registers with * are asynchronously reset to 0 at start-up so that GPIO pins are inactive. Reprinted with permission from Table 52 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

five most important registers for interacting with the GPIO pins are, as described above, `input_val`, `input_en`, `output_en`, `output_val`, and `iof_en` at offsets of 0x0, 0x4, 0x8, 0xC, and 0x38 from the base address. Each register is 32 bits wide and could control up to 32 GPIOs, but only 19 GPIOs are physically present on this chip.

To read GPIO *n*, a program sets bit *n* of the `input_en` (input enable) register and then reads the `input_val` (input value) register and looks at bit *n*. Similarly, to drive GPIO *n*, a program sets bit *n* of the `output_en` (output enable) register and then writes the desired value to bit *n* of the `output_val` (output value) register. In both cases, bit *n* of

In the context of bit manipulation, "setting" means to write 1 to a bit and "clearing" means to write 0 to a bit.

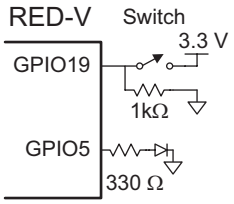


Figure e9.5 LED output on GPIO pin 5 and switch input from GPIO pin 19

the `iof_en` register must be cleared to ensure that the pin is driven by the GPIO controller instead of other hardware on the chip.

Code Example e9.1 illustrates a simple program that reads the value of the switch connected to GPIO19 and accordingly turns ON or OFF the on-board LED connected to GPIO5. The hardware setup is shown in Figure e9.5. To access the memory-mapped I/O, it first declares pointers to the five registers at the addresses mentioned above. Each pointer is of type `uint32_t*` because the registers contain unsigned 32-bit values. The program writes a 1 to bit 19 of the `input_en` register and a 1 to bit 5 of the `output_en` register to configure GPIO pin 19 as an input and GPIO pin 5 as an output. Notice how we use the shift operation (`1 << 19`) to set a 1 in bit 19 and OR it with the existing contents of the enable register to turn on that bit without affecting other bits that might already be turned on. Then, we write a 0 to bits 5 and 19 in the `iof_en` register to ensure that the pins are driven by the GPIO controller. To write a 0 to a bit, we AND `iof_en` with 1's in every position except that bit so that the desired bit is forced low and the other bits are not affected. Next, the program repeatedly reads the input pin and writes the output pin. To read the input pin, the program reads the `input_val` register, right-shifts the value by 19 (to move pin 19's value into bit 0), and performs a bitwise `AND` with `0x1` to retain only bit 0, leaving a single 0 or 1 corresponding to the value originally in bit 19. To write a high value to a bit of the `output_val` register, we use the `OR` operation, as we did to turn on a bit in the enable registers. To write a 0 to a bit in the `output_val` register, we use the same approach as described above for clearing bits in the `iof_en` register.

Code Example e9.1 SETTING GPIO OUTPUT BASED ON SWITCH INPUT

```
#include <stdint.h>
int main(void) {
    volatile uint32_t *input_val  = (uint32_t*)0x10012000;
    volatile uint32_t *input_en   = (uint32_t*)0x10012004;
    volatile uint32_t *output_en  = (uint32_t*)0x10012008;
    volatile uint32_t *output_val = (uint32_t*)0x1001200C;
    volatile uint32_t *iof_en     = (uint32_t*)0x10012038;
    int val;

    *iof_en    &= ~(1 << 19);              // Pin 19 is a GPIO
    *input_en  |=  (1 << 19);              // Pin 19 is an input
    *iof_en    &= ~(1 << 5);               // Pin 5 is a GPIO
    *output_en |=  (1 << 5);               // Pin 5 is an output
    while (1) {
        val = (*input_val >> 19) & 1;      // Read value on pin 19
        if (val) *output_val |= (1 << 5);  // Turn ON pin 5
        else     *output_val &= ~(1 << 5); // TURN OFF pin 5
    }
}
```

Other GPIO Registers

Table e9.2 listed several other GPIO control registers of interest, particularly the pin drive strength (`ds`), internal pull-up enable (`pue`), and I/O function (`iof_sel` and `iof_en`) registers.

The `ds` register controls each pin's maximum output current. The default value (0) configures I_{OL}/I_{OH} as 15 to 16 mA, while setting a pin's `ds` to 1 increases that pin's output current modestly to 21 mA, which might be helpful to drive a brighter LED.

The `pue` register configures an internal pull-up resistor. Figure e9.4 showed an example of an external *pull-down* resistor. If the power and ground connections on the switch were reversed, the resistor would then be a *pull-up* resistor that drives the pin to 1 when the switch is not connected. In that case, when the switch was pressed, the pin would drop to 0. To save money and circuit board space, many microcontrollers contain internal pull-up resistors that can optionally be enabled in software. Writing a 1 to a bit of the `pue` register activates the internal pull-up resistor for the corresponding GPIO pin. According to Table 4.2 of the FE310-G002 datasheet, the pull-up current is 85 µA when the pin is at 0 V. Hence, the effective pull-up resistance is 3.3 V/85 µA = 39 kΩ (V/I = R).

As shown in Table e9.3, most GPIO pins can also perform a special function, such as acting as a serial port or a pulse-width modulation (PWM) output. We discuss these functions in detail later in this chapter. The `iof_sel` and `iof_en` registers together determine whether each pin is acting as a GPIO or as a special function. When `iof_en` is 0 (the default), the pin acts as a GPIO. When it is 1, it takes on the special function. The special function is chosen from Table e9.3 based on the `iof_sel` bit for that pin. For example, to use GPIO11 to generate a pulse-width modulated waveform, set bit 11 of `iof_sel` and `iof_en` to 1. Then, use the PWM registers to control the output. `iof_en` is mapped to address 0x10012038 and `iof_sel` to 0x1001203C. Table e9.3 lists 32 GPIOs; however, remember that the RED-V boards only include 19 GPIOs: GPIOs 0 to 5, 9 to 13, and 16 to 23.

9.3.4 Device Drivers

As we saw in Code Example e9.1, programmers can manipulate I/O devices directly by reading or writing the memory-mapped I/O registers. However, it is better programming practice to call functions that access the memory-mapped I/O. These functions are called *device drivers*. Some of the benefits of using device drivers include:

- The code is easier to read when it involves a clearly named function call rather than a write to bit fields at an obscure memory address.
- Somebody who is familiar with the deep workings of the I/O devices can write the device driver and casual users can call it without having to understand the details.

Table e9.3 GPIO pins special functions map

GPIO Number	IOF0	IOF1
0		PWM0_PWM0
1		PWM0_PWM1
2	SPI1_CS0	PWM0_PWM2
3	SPI1_DQ0	PWM0_PWM3
4	SPI1_DQ1	
5	SPI1_SCK	
6	SPI1_DQ2	
7	SPI1_DQ3	
8	SPI1_CS1	
9	SPI1_CS2	
10	SPI1_CS3	PWM2_PWM0
11		PWM2_PWM1
12	I2C0_SDA	PWM2_PWM2
13	I2C0_SCL	PWM2_PWM3
14		
15		
16	UART0_RX	
17	UART0_TX	
18	UART1_TX	
19		PWM1_PWM1
20		PWM1_PWM0
21		PWM1_PWM2
22		PWM1_PWM3
23	UART1_RX	
24		
25		
26	SPI2_CS0	
27	SPI2_DQ0	
28	SPI2_DQ1	
29	SPI2_SCK	
30	SPI2_DQ2	
31	SPI2_DQ3	

- The code is easier to port to another processor with different memory mapping or I/O devices because only the device driver must change.
- If the device driver is part of the operating system (OS), the OS can control access to physical devices shared among multiple programs running on the system and can manage security (e.g., so a malicious program can't read the keyboard while you are typing your password into a web browser).

This chapter will develop a simple device driver called EasyREDVIO to access FE310 peripherals so that you can understand what is happening under the hood in a device driver. To access all features of the FE310, users may prefer the Freedom Metal environment, which provides convenient software interfaces for controlling SiFive Core IP features and peripheral devices. Freedom Metal is powerful since it is written in such a way that its API will work on any device that has a Freedom Metal board support package (BSP). A BSP is a software package containing drivers and other commonly used routines. SiFive also provides the Freedom E software developer kit (SDK) and Freedom Studio, which allow users to develop software for any SiFive core.

EasyREDVIO and the code examples in this chapter can be downloaded from the textbook website (see the Preface). More information about Freedom Metal and documentation can be found at https://github.com/sifive/freedom-metal.

Example e9.2 DEVICE DRIVERS IN C

Accessing and modifying the values for memory-mapped I/O is accomplished by reading or writing to memory addresses. In assembly, this is done using `lw` and `sw` instructions. As illustrated in Code Example e9.2, C can do the same thing with pointers, but it is tedious and error-prone to declare pointers for every memory-mapped I/O register. A more natural way to describe and control memory-mapped I/O in C is using structures.

As discussed in Section C.8.5 in the appendix, structures in C are a way to group a collection of different data types into a single unit. Using structures in the context of memory-mapped registers allows communication with the I/O device using the name of a given register or field as opposed to a memory address. A C program can declare a structure for a memory-mapped peripheral, listing the registers in the order they appear in the memory map. It can then declare a pointer to such a structure and access the peripheral via the structure pointer.

Start the EasyREDVIO library by writing `pinMode`, `digitalRead`, and `digitalWrite` functions to configure a pin's direction and read or write it.

- The `pinMode` function takes two inputs: the pin number and the mode. For example, `pinMode(5, INPUT)` sets GPIO pin 5 as an input, and `pinMode(17, OUTPUT)` sets GPIO pin 17 as an output.
- `digitalRead` takes one input, the pin number, and returns the value of that pin. For example, `digitalRead(19)` reads the value of GPIO19.
- `digitalWrite` takes two inputs: the pin number and the value. For example, `digitalWrite(3, 1)` writes 1 to GPIO pin 3, and `digitalWrite(5, 0)` writes 0 to GPIO pin 5.

After writing these functions, write a C program that uses these functions to read the three switches and turn on the corresponding LEDs, using the hardware in Figure e9.4.

Solution The EasyREDVIO code is given below. The functions must choose which registers and bits within those registers to access. For example, to

configure a pin as an input, `pinMode` must set that pin's bit in `input_en` and clear that pin's bit in `output_en`. `digitalWrite` handles writing either 1 or 0 by writing to `output_val`. `digitalRead` reads the desired bit of `input_val`.

The GPIO structure (struct) specifies the 32-bit registers by name. Two `define` statements then specify the base address of the GPIO (`GPIO0_BASE`) and instantiate a pointer of type GPIO located at that base address. Each of the 32-bit variables in the structure are then located in memory in ascending order from that base address.

Code Example e9.2 GPIO FOR SWITCHES AND LEDS

```
// EasyREDVIO.h
// Joshua Brake, David Harris, and Sarah Harris, 7 October 2020

#include <stdint.h>

#define INPUT  0
#define OUTPUT 1

// Define statements to map Arduino pin names to FE310 GPIO pin number according to Figure e9.2
#define D0  16
#define D1  17
#define D2  18
#define D3  19
#define D4  20
#define D5  21
#define D6  22
#define D7  23
#define D8  0
#define D9  1
#define D10 2
#define D11 3
#define D12 4
#define D13 5
#define D15 9
#define D16 10
#define D17 11
#define D18 12
#define D19 13

// Declare a GPIO structure defining the GPIO registers in the order they appear in Table e9.2
typedef struct {
    volatile uint32_t  input_val;     // (GPIO offset 0x00) Pin value
    volatile uint32_t  input_en;      // (GPIO offset 0x04) Pin input enable*
    volatile uint32_t  output_en;     // (GPIO offset 0x08) Pin output enable*
    volatile uint32_t  output_val;    // (GPIO offset 0x0C) Output value
    volatile uint32_t  pue;           // (GPIO offset 0x10) Internal pull-up enable*
    volatile uint32_t  ds;            // (GPIO offset 0x14) Pin drive strength
    volatile uint32_t  rise_ie;       // (GPIO offset 0x18) Rise interrupt enable
    volatile uint32_t  rise_ip;       // (GPIO offset 0x1C) Rise interrupt pending
    volatile uint32_t  fall_ie;       // (GPIO offset 0x20) Fall interrupt enable
    volatile uint32_t  fall_ip;       // (GPIO offset 0x24) Fall interrupt pending
    volatile uint32_t  high_ie;       // (GPIO offset 0x28) High interrupt enable
    volatile uint32_t  high_ip;       // (GPIO offset 0x2C) High interrupt pending
    volatile uint32_t  low_ie;        // (GPIO offset 0x30) Low interrupt enable
    volatile uint32_t  low_ip;        // (GPIO offset 0x34) Low interrupt pending
    volatile uint32_t  iof_en;        // (GPIO offset 0x38) HW-Driven functions enable
    volatile uint32_t  iof_sel;       // (GPIO offset 0x3C) HW-Driven functions selection
    volatile uint32_t  out_xor;       // (GPIO offset 0x40) Output XOR (invert)
    // Registers marked with * are asynchronously reset to 0 at startup
} GPIO;
```

```
// Define the base address of the GPIO registers (see Table e9.1) and a pointer to this
// structure
// The 0x…U notation in 0x10012000U indicates an unsigned hexadecimal number
#define GPIO0_BASE  (0x10012000U)
#define GPIO0 ((GPIO*) GPIO0_BASE)

// To access the members of the structure, the member-access operator -> is used.

void pinMode(int gpio_pin, int function) {
    switch(function) {
        case INPUT:
            GPIO0->input_en   |=  (1 << gpio_pin);  // Sets a pin as an input
            GPIO0->output_en  &= ~(1 << gpio_pin);  // Clear output_en bit
            GPIO0->iof_en     &= ~(1 << gpio_pin);  // Disable IOF
            break;
        case OUTPUT:
            GPIO0->output_en  |=  (1 << gpio_pin);  // Set pin as an output
            GPIO0->input_en   &= ~(1 << gpio_pin);  // Clear input_en bit
            GPIO0->iof_en     &= ~(1 << gpio_pin);  // Disable IOF
            break;
    }
}

void digitalWrite(int gpio_pin, int val) {
    if (val) GPIO0->output_val |=  (1 << gpio_pin);
    else     GPIO0->output_val &= ~(1 << gpio_pin);
}

int digitalRead(int gpio_pin) {
    return (GPIO0->input_val >> gpio_pin) & 0x1;
}

// The program below reads switches and writes LEDs. It sets pins 2 to 4 as inputs (for the
// switches) and pins 7 to 9 as outputs (for the LEDs). It then continuously reads the
// switches and writes their values to the corresponding LEDs.

#include "EasyREDVIO.h"
int main(void) {
  // Set GPIO 4:2 as inputs
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(4, INPUT);
  // Set GPIO 10:8 as outputs
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  while (1) { // Read each switch and write corresponding LED
    digitalWrite(8,  digitalRead(2));
    digitalWrite(9,  digitalRead(3));
    digitalWrite(10, digitalRead(4));
  }
}
```

9.3.5 Serial I/O

A microcontroller can send multiple bits to a peripheral device by using multiple wires or by sending multiple bits in series over a single wire. The former is called *parallel I/O* and the latter is called *serial I/O*. Serial I/O is popular, especially when pins are limited, because it uses few wires and is fast enough for many applications. Indeed, it is so popular that many standards for serial I/O have been established

and microcontrollers offer dedicated hardware to easily send data via these standards. This section describes the SPI and UART standard serial interfaces.

Other common serial standards include inter-integrated circuit (I^2C), universal serial bus (USB), and Ethernet. I^2C (pronounced "I squared C") is a 2-wire interface with a clock and a bidirectional data pin; it is used in a fashion similar to SPI. USB and Ethernet are more complex, high-performance standards. The FE310 supports SPI, UART, and I^2C via on-board specialized peripherals.

The terms *master/slave* used to be common (instead of *controller/peripheral*), but they are now outdated. *Serial data out (SDO)* or *controller-out peripheral-in (COPI)* is now used in place of *master-out slave-in (MOSI)*. *Serial data in (SDI) or controller-in peripheral-out (CIPO)* is now used in place of *master-in slave-out (MISO)*.

Serial Peripheral Interface (SPI)

SPI (pronounced "S-P-I") is a simple synchronous serial protocol that is easy to use and relatively fast. The physical interface consists of three pins: serial clock (SCK), serial data out (SDO), and serial data in (SDI). SPI connects a *controller* device to a *peripheral* device, as shown in Figure e9.6(a). The controller produces the clock. It initiates communication by sending clock pulses on SCK. The controller sends data from its SDO pin to the peripheral's SDI pin one bit per cycle, starting with the most significant bit. The peripheral may simultaneously respond with its SDO pin back to the controller's SDI pin. Figure e9.6(b) shows the SPI waveforms for an 8-bit data transmission. Bits change on the falling edge of SCK and are stable to sample on the rising edge. The SPI interface may also send an active-low chip enable to alert the receiver that data is coming.

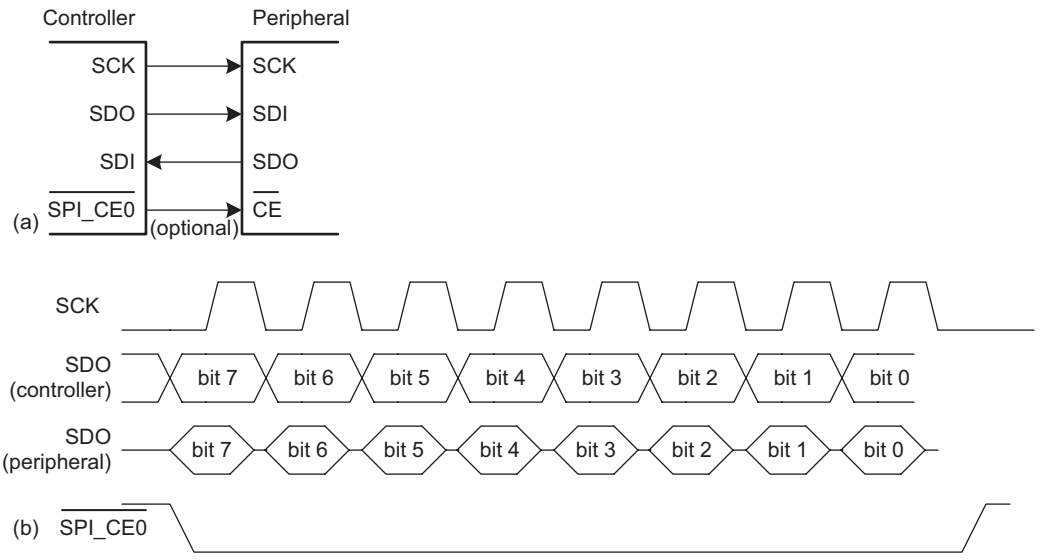


Figure e9.6 **SPI configuration: (a) SPI controller-peripheral connection diagram, (b) Example SPI data signals**

The FE310 has three SPI controller ports, but only two (SPI1 and SPI2) are available to the user. The remaining SPI controller port, SPI0, is used to communicate with external flash memory for program and data storage. This section describes the SPI1 controller port, which is accessible using GPIO pins 5:2. The SPI2 controller port is identical except that it is connected to different GPIO pins and its control registers are located at different memory addresses. To use pins for SPI rather than GPIO, their `iof_sel` bits should be set to 0 to select the SPI1 function and their `iof_en` bits should be set to 1 to give the SPI controller access to the pins. When the FE310 writes to the SPI `txdata` register, the data is transmitted serially to the peripheral. Simultaneously, data received from the peripheral is collected and the FE310 can read it from `rxdata` when the transfer is complete.

The term *port* refers to a pin or a group of associated pins.

SPI ports on a microcontroller normally offer a variety of configuration options to match the requirements of peripheral devices. When designing an interface to communicate with a particular peripheral device, the controller must be configured properly to ensure that the data being transmitted via the link is properly interpreted.

Two common configuration parameters are clock polarity (CPOL) and clock phase (CPHA). CPOL sets the level of the clock when it is idle and CPHA sets the clock edge when data (SDO and SDI) is sampled (and changed). If CPOL = 1, SCK remains high (1) when data is not being transmitted; if CPOL = 0, SCK remains low (0) when idle. If CPHA = 0, data are sampled on the leading edge (and change on the trailing edge) of SCK; if CPHA = 1, data are sampled on the trailing edge (and change on the leading edge) of SCK. The edge on which data changes is also referred to as the *shifting edge* because the underlying hardware is usually a shift register. Figure e9.7 shows the four possible combinations of CPHA and CPOL. The example from Figure e9.6 shows CPOL = 0 and CPHA = 0.

SPI always sends data in both directions on each transfer. If the system only needs unidirectional communication, it can ignore the unwanted data. For example, if the controller only needs to send data to the peripheral, the byte received from the peripheral can be ignored. If the controller only needs to receive data from the peripheral, it must still trigger the SPI communication by sending an arbitrary byte that the peripheral will ignore. It can then read the data received from the peripheral. The SPI clock (SCK) only toggles while the controller is transmitting data.

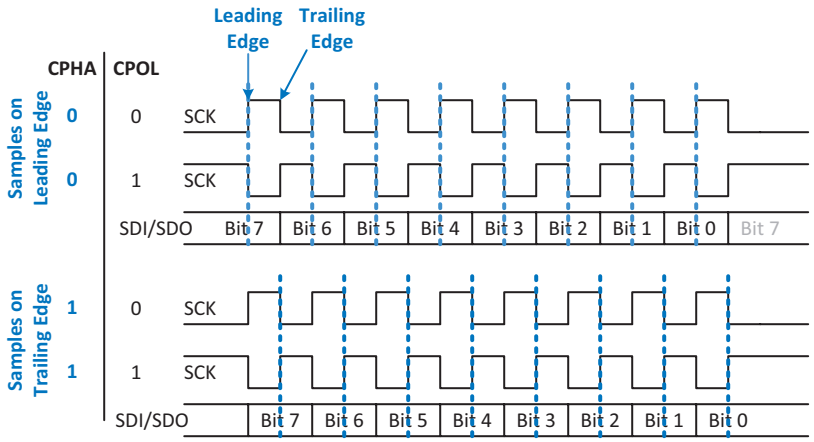


Figure e9.7 Timing diagram and configurations for SPI peripherals

Table e9.4 Memory map of SPI registers

Offset	Name	Description
0x00	sckdiv	Serial clock divisor
0x04	sckmode	Serial clock mode
0x08	Reserved	
0x0C	Reserved	
0x10	csid	Chip select ID
0x14	csdef	Chip select default
0x18	csmode	Chip select mode
0x1C	Reserved	
0x20	Reserved	
0x24	Reserved	
0x28	delay0	Delay control 0
0x2C	delay1	Delay control 1
0x30	Reserved	
0x34	Reserved	
0x38	Reserved	
0x3C	Reserved	
0x40	fmt	Frame format
0x44	Reserved	
0x48	txdata	Tx FIFO Data
0x4C	rxdata	Rx FIFO data
0x50	txmark	Tx FIFO watermark
0x54	rxmark	Rx FIFO watermark
0x58	Reserved	
0x5C	Reserved	
0x60	fctrl	SPI flash interface control*
0x64	ffmt	SPI flash instruction format*
0x68	Reserved	
0x6C	Reserved	
0x70	ie	SPI interrupt enable
0x74	ip	SPI interrupt pending

Reprinted with permission from Table 65 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

Table e9.4 shows the control registers associated with SPI1, and Table e9.5 shows the fields of the key registers. `sckdiv` (see Table e9.4) configures the SPI clock frequency by specifying a divisor (`div`) for the selected input peripheral clock—on the RED-V board, the peripheral clock's default frequency is 16 MHz. The frequency of the SPI clock is given by $f_{sck} = \frac{f_{in}}{2(div + 1)}$. For example, if `div` = 15, then the serial clock is $f_{sck} = \frac{16\,\text{MHz}}{2(15 + 1)} = 500\,\text{kHz}$. If the frequency is too high (>~1 MHz on a breadboard or tens of MHz on an unterminated printed circuit board), the SPI connection may become unreliable due to reflections, crosstalk, or other signal integrity issues.

Table e9.5 SPI register bitfields

Serial Clock Divisor Register (sckdiv)				
Register Offset		0x0		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[11:0]	div	RW	0x3	Divisor for serial clock. div_width bits wide.
[31:12]	Reserved			

Serial Clock Mode Register (sckmode)				
Register Offset		0x4		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	pha	RW	0x0	Serial clock phase
1	pol	RW	0x0	Serial clock polarity
[31:2]	Reserved			

Transmit Data Register (txdata)				
Register Offset		0x48		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	data	RW	0x0	Transmit data
[30:8]	Reserved			
31	full	RO	X	FIFO full flag

Receive Data Register (rxdata)				
Register Offset		0x4C		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	data	RO	X	Received data
[30:8]	Reserved			
31	empty	RW	X	FIFO empty flag

Configuration registers have many unused or "reserved" bits. These bits might be used in a future version of the chip, so they should not be written lest they cause unintended consequences in the future.

`sckmode` controls the phase and polarity of the clock. `sckmode` uses only the two least significant bits. Bit 0 is CPHA and bit 1 is CPOL.

`txdata` is written to transmit a byte over the SPI channel, and `rxdata` is read to get the received byte. Only the least significant byte (LSB) written to `txdata` is transmitted. The SPI instances on the FE310 have 8-entry first-in-first-out (FIFO) buffers on both the transmit and receive data registers. This means that when data is written to the `txdata` register, it is placed in the FIFO buffer and the hardware within the SPI peripheral takes care of sending it out. The most significant bit (msb) of the `txdata` register is a flag bit called `full`, which is 1 when the FIFO is full and cannot receive any more data.

Care must be taken when reading data in from the FE310 SPI `rxdata` register. The SPI controller is designed such that the data in the register is removed from the receive FIFO when the register is read. To check if the `rxdata` register has valid data, the register should be read

once and then the `empty` bit should be checked to determine if the data is valid. The programmer should take care to avoid reading the register more than once for each byte, as this will result in lost data.

The registers `csid`, `csdef`, and `csmode` are optionally used to control parameters related to the control and configuration of the chip select line. Alternatively, the chip select pin can be configured as a GPIO output pin and controlled in software through `digitalWrite`.

Some SPI registers pack multiple small fields of information into a single 32-bit word. In C, we can declare the number of bits of each field with a colon and number as part of a bitfield structure. Example e9.3 shows how to use bitfields and structures to define these registers.

Example e9.3 SENDING AND RECEIVING BYTES OVER SPI

Design a system to communicate between a FE310 controller and an FPGA peripheral over SPI. Sketch a schematic of the interface. Write the C code for the FE310 to send the character "A" and receive a character back. Write HDL code for an SPI peripheral on the FPGA. How could the peripheral be simplified if it only needs to receive data?

Solution Figure e9.8 shows the connection between the FE310 controller and the FPGA peripheral using SPI1. The pin numbers are obtained from the component datasheets (e.g., Table e9.3 for the FE310). Notice that both the pin numbers and signal names are shown on the diagram to indicate both the physical and logical connectivity. When the SPI connection is enabled, these pins cannot be used for GPIO.

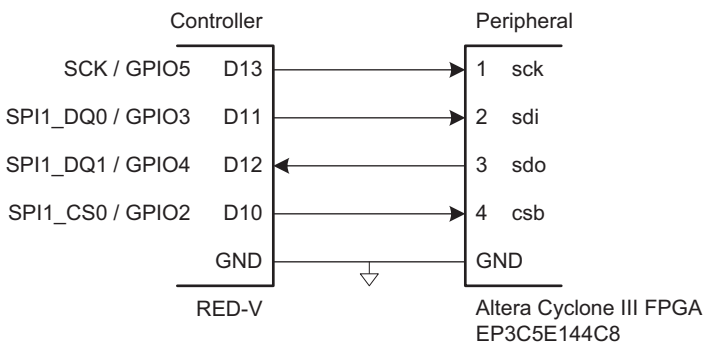


Figure e9.8 RED-V to Altera Cyclone FPGA connection diagram

The following code from EasyREDVIO.h is used to initialize the SPI channel and to send and receive a character. The file first declares the SPI bitfields and memory map. The `pinMode` function is generalized to support I/O functions as well as inputs and outputs. The function `spiSendReceive` completes a full SPI transaction sending and receiving a single byte. It initially checks to make sure that the transmit FIFO is not full and can accept another entry. If yes, it writes the character to the transmit FIFO

to be shifted out. After transmitting, the `rxdata` register is read. Here, care must be taken because the empty flag bit of the `rxdata` register is updated whenever the register is read. So, the entire 32-bit `rxdata` register should be read. Then, after checking that the empty flag is not set (i.e., the data is valid), the received byte is returned.

Code Example e9.3 SPI FUNCTIONS

```
////////////////////////////////////////////////////////////////////////////////
// SPI Registers
////////////////////////////////////////////////////////////////////////////////
typedef struct {
    volatile uint32_t     div            :     12;   // Clock divisor
    volatile uint32_t                    :     20;
} sckdiv_bits;

typedef struct {
    volatile uint32_t     pha            :     1;    // Serial clock phase
    volatile uint32_t     pol            :     1;    // Serial clock polarity
    volatile uint32_t                    :     30;
} sckmode_bits;

...

typedef struct {
    volatile uint32_t     data           :     8;    // Transmit data
    volatile uint32_t                    :     23;
    volatile uint32_t     full           :     1;    // FIFO full flag
} txdata_bits;

typedef struct {
    volatile uint32_t     data           :     8;    // Received data
    volatile uint32_t                    :     23;
    volatile uint32_t     empty          :     1;    // FIFO empty flag
} rxdata_bits;

// Pin modes
#define INPUT  0
#define OUTPUT 1
#define GPIO_IOF0 2
#define GPIO_IOF1 3

void pinMode(int gpio_pin, int function) {
    switch(function) {
        case INPUT:
            GPIO0->input_en   |=  (1 << gpio_pin);   // Set a pin as an input
            GPIO0->iof_en     &= ~(1 << gpio_pin);   // Disable IOF
            break;
        case OUTPUT:
            GPIO0->output_en  |=  (1 << gpio_pin);   // Set pin as an output
            GPIO0->iof_en     &= ~(1 << gpio_pin);   // Disable IOF
            break;
        case GPIO_IOF0:
            GPIO0->iof_en     |=  (1 << gpio_pin);   // Enable IOF
            GPIO0->iof_sel    &= ~(1 << gpio_pin);   // IO Function 0
            break;
        case GPIO_IOF1:
            GPIO0->iof_en     |= (1 << gpio_pin);    // Enable IOF
            GPIO0->iof_sel    |= (1 << gpio_pin);    // IO Function 1
            break;
    }
}
```

```
void spiInit(uint32_t clkdivide, uint32_t cpol, uint32_t cpha) {

    // Initially assigning SPI pins (GPIO 2-5) to HW I/O function 0
    pinMode(3, GPIO_IOF0); // SDO
    pinMode(4, GPIO_IOF0); // SDI
    pinMode(5, GPIO_IOF0); // SCK

    digitalWrite(2, 1);    // make sure CS0 doesn't pulse low
    pinMode(2, OUTPUT);    // CS0 is manually controlled

    SPI1->sckdiv.div = clkdivide; // Set the clock divisor

    SPI1->sckmode.pol = cpol;     // Set the polarity
    SPI1->sckmode.pha = cpha;     // Set the phase
}

/* Transmits a character (1 byte) over SPI and returns the received character.
 *    send: the character to send over SPI
 *    return value: the character received over SPI */
uint8_t spiSendReceive(uint8_t send) {
    while(SPI1->txdata.full); // Wait until transmit FIFO is ready for new data
    SPI1->txdata.data = send;  // Transmit the character over SPI

    rxdata_bits rxdata;
    while (1) {
        rxdata = SPI1->rxdata; // Read the rxdata register EXACTLY once
        if (!rxdata.empty) {   // If the empty bit was not set, return the data
            return (uint8_t)rxdata.data;
        }
    }
}
```

The C code in Code Example e9.4 initializes the SPI and then sends and receives a character. Using the formula $f_{clk} = \frac{f_{in}}{2(div + 1)}$, where f_{in} is the 16 MHz `coreclk`, it sets the SPI clock to 500 kHz.

Code Example e9.4 SPI FUNCTIONS

```
#include "EasyREDVIO.h"

int main(void) {
  uint8_t volatile received;

  // Initialize the SPI
  // Clock divisor of div = 15, CPOL = 0, CPHA = 0
  spiInit(15, 0, 0);

  digitalWrite(2, 0);             // enable the peripheral (chip select = 0), if necessary
  received = spiSendReceive('A'); // Send letter A and receive byte
  digitalWrite(2, 1);             // turn off chip enable
}
```

If the peripheral needs to receive data only from the controller, it is a simple shift register, as shown in HDL Example e9.1. On each rising `sck` edge, a new `sdi` bit is shifted into the shift register, starting with the data's most significant bit. After eight clock cycles, the entire byte has been read into the shift register.

HDL Example e9.1 HDL FOR SPI PERIPHERAL (RECEIVER ONLY)

```
module spi_peripheral_receive_only(input  logic       sck, // From controller
                                   input  logic       sdi, // From controller
                                   output logic [7:0] q);  // Data received
  always_ff @(posedge sck)
    q <= {q[6:0], sdi}; // shift register
endmodule
```

HDL Example e9.2 gives the SystemVerilog code for an SPI peripheral that can both send and receive data (i.e., an SPI transceiver), and Figure e9.9 shows its block diagram and timing with CPHA = CPOL = 0. The main component is still a shift register, shown on the right of Figure e9.9. The shift register parallel loads the byte to send (`d[7:0]`) into the shift register and then shifts out this data on `sdo` while it shifts in data transmitted from the controller (`t[7:0]`) on `sdi`. A counter, `cnt`, keeps track of how many bits have been sent/received. When `sck` is idle, `cnt` = 0 and the most significant bit of `d` (`d[7]`) sits on the `sdo` wire. One subtlety is that `sdo` can only change on the falling clock edge, so the `sdo` output (which is the most significant bit of the shift register, `q[7]`, is delayed by half a clock cycle by the negative-edge triggered qdelayed register on the bottom left of Figure e9.9.

HDL Example e9.2 HDL FOR SPI PERIPHERAL

```
module spi_peripheral(input  logic       sck,   // From controller
                      input  logic       sdi,   // From controller
                      output logic       sdo,   // To controller
                      input  logic       reset, // System reset
                      input  logic [7:0] d,     // Data to send
                      output logic [7:0] q);    // Data received
  logic [2:0] cnt;
  logic  qdelayed;

  // 3-bit counter tracks when full byte is transmitted
  always_ff @(negedge sck, posedge reset)
    if (reset) cnt = 0;
    else       cnt = cnt + 3'b1;

  // Loadable shift register
  // Loads d at the start, shifts sdi into bottom on each step
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], sdi} : {q[6:0], sdi};

  // Align sdo to falling edge of sck
  // Load d at the start
  always_ff @(negedge sck)
    qdelayed = q[7];

  assign sdo = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

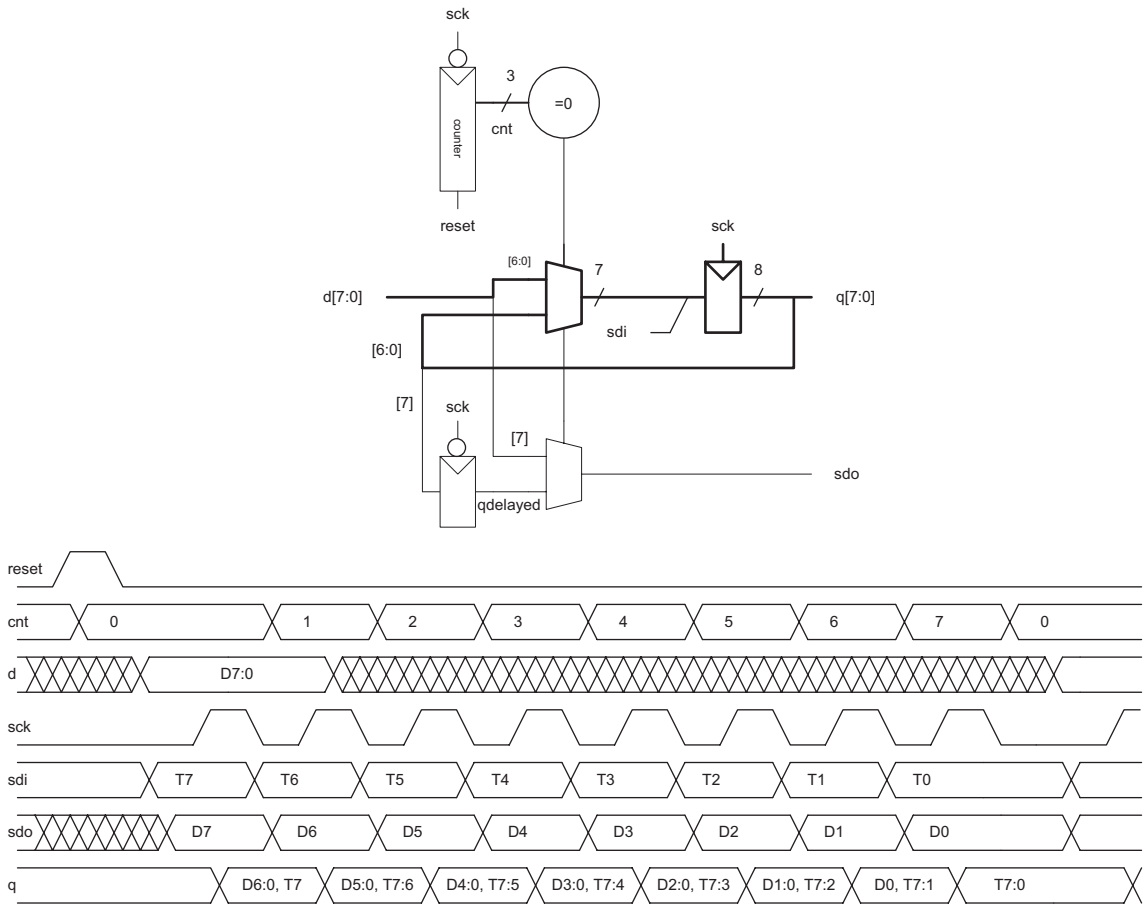


Figure e9.9 **Block and timing diagram for SPI peripheral on FPGA**

Universal Asynchronous Receiver/Transmitter (UART)

A UART (pronounced "you-art") is a serial I/O peripheral that communicates between two systems without sending a clock. Instead, the systems must agree in advance about what data rate to use and must each locally generate their own clocks. Hence, the transmission is asynchronous because the clocks are not synchronized. Although these system clocks may have a small frequency error and an unknown phase relationship, the UART manages reliable asynchronous communication. UARTs are used in protocols such as RS-232 and RS-485. For example, old computer serial ports use the RS-232C standard, introduced in 1969 by the Electronics Industries Associations. The standard originally envisioned connecting *data terminal equipment* (DTE) such as a mainframe

computer to *data communication equipment* (DCE) such as a modem. Although a UART is relatively slow compared with SPI and prone to misconfiguration issues, the standards have been around for so long that they remain important today.

Figure e9.10(a) shows an asynchronous serial link. The DTE sends data to the DCE over the TX line and receives data back over the RX line. Figure e9.10(b) shows one of these lines sending a character at a data rate of 9600 baud. The lines idle at a logic "1" when not in use. Each character is sent as a start bit (0), 7 or 8 data bits, an optional parity bit, and one or more stop bits (1's). Most typically, start and stop bits and 8 bits of data are sent. The UART detects the falling transition from idle to start to lock on to the transmission at the appropriate time. Although seven data bits is sufficient to send an ASCII character, eight bits are normally used because they can convey an arbitrary byte of data.

The optional parity bit allows the system to detect if a bit was corrupted during transmission. It can be configured as *even* or *odd*; *even parity* means that the parity bit is chosen such that the total collection of data and parity has an even number of 1's. In other words, the parity bit is the XOR of the data bits. The receiver can then check if an even number of 1's was received and signal an error if not. *Odd parity* is the reverse.

A common choice is 1 start bit, 8 data bits, no parity, and 1 stop bit, making a total of 10 symbols to convey an 8-bit character of information. Hence, signaling rates are referred to in units of baud rather than bits/sec. For example, 9600 baud indicates 9600 symbols/sec, or 960 characters/second. Both the transmitter and receiver must be configured for the appropriate baud rate and number of data, parity, and stop bits or the data will be garbled. This is a hassle, especially for nontechnical users, which is one of the reasons that USB has replaced UARTs in personal computer systems.

Typical baud rates include 300, 1200, 2400, 9600, 14400, 19200, 38400, 57600, and 115200. The lower rates were used in the 1970's and 1980's for modems that sent data over the phone lines as a series of tones. In contemporary systems, 9600 and 115200 are two of the most

Baud rate gives the *signaling rate*, measured in symbols per second, whereas bit rate gives the *data rate*, measured in bits per second. In a simple system like SPI, where each symbol is a data bit, the baud rate is equal to the bit rate. UARTs and some other signaling conventions require overhead bits in addition to the data. For example, a UART that adds start and stop bits for each 8 bits of data (i.e., 10 symbols per 8 bits of data) and operates at a baud rate of 9600 has a bit rate of (9600 symbols/second) × (8 bits/10 symbols) = 7680 bits/second = 960 characters/second.

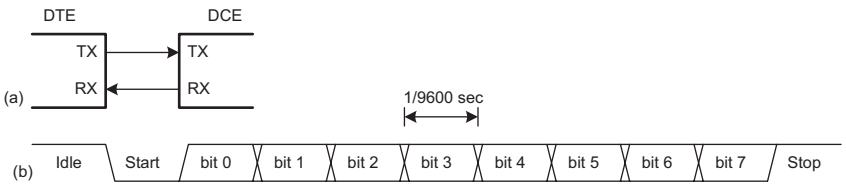


Figure e9.10 Asynchronous serial link

In the 1950's through 1970's, early hackers calling themselves *phone phreaks* learned to control the phone company switches by whistling appropriate tones. A 2600 Hz tone produced by a toy whistle from a Cap'n Crunch cereal box (Figure e9.11). could be exploited to place free long-distance and international calls.

Figure e9.11 Cap'n Crunch Bosun Whistle
(Photograph by Evrim Sen, reprinted with permission.)

common baud rates; 9600 is encountered where speed does not matter, and 115200 is the fastest standard rate, though still slow compared with other modern serial I/O standards.

The RS-232 standard defines several additional signals. The request to send (RTS) and clear to send (CTS) signals can be used for *hardware handshaking*. They can be operated in either of two modes: flow control or simplex. In *flow control* mode, the DTE clears RTS to 0 when it is ready to accept data from the DCE. Likewise, the DCE clears CTS to 0 when it is ready to receive data from the DTE. Some datasheets use an overbar to indicate that they are active-low. In the older *simplex* mode, the DTE clears RTS to 0 when it is ready to transmit. The DCE replies by clearing CTS when it is ready to receive the transmission.

Some systems, especially those connected over a telephone line, also used data terminal ready (DTR), data carrier detect (DCD), data set ready (DSR), and ring indicator (RI) signals to indicate when equipment is connected to the line. These signals still show up in some connectors.

The original RS-232 standard recommended a massive 25-pin DB-25 connector, but PCs streamlined it to a male 9-pin DE-9 connector with the pinout shown in Figure e9.12(a). The cable wires normally connect straight across, as shown in Figure e9.12(b). However, when directly connecting two DTEs, a *null modem* cable shown in Figure e9.12(c) may be needed to swap RX and TX and complete the handshaking. As a final insult, some connectors are male and some are female. In summary, it can take a large box of cables and a certain amount of guesswork to connect two systems over RS-232, again explaining the shift to USB. Fortunately, embedded systems typically use a simplified 3- or 5-wire setup consisting of GND, TX, RX, and possibly RTS and CTS.

Handshaking refers to the negotiation between two systems. Typically, one system signals that it is ready to send or receive data and the other system acknowledges that request.

RS-232 represents a 0 electrically with 3 to 15 V and a 1 with −3 to −15 V; this is called *bipolar* signaling. A transceiver converts the digital logic levels of the UART to the positive and negative levels expected by RS-232 and also provides electrostatic discharge protection to protect the serial port from getting zapped when the user plugs in a cable. The MAX3232E is a popular transceiver compatible with both 3.3 and 5 V digital logic. It contains a charge pump that, in conjunction with external capacitors, generates ±5 V outputs from a single low-voltage power supply. Some serial peripherals intended for embedded systems omit the transceiver and just use 0 V for a 0 and 3.3 or 5 V for a 1; check the datasheet!

The FE310 has two UARTs, named UART0 and UART1. UART0 can be configured to operate on pins 16 and 17; UART1 operates on pins 18 and 23. To use these pins as a UART instead of as GPIOs, their corresponding `iof_sel` bits should be set to 0 (to select `IOF0`) and

`iof_en` bits set to 1 to enable peripheral control. As with SPI, the FE310 must first configure the port. Unlike SPI, reading and writing can occur independently because either system may transmit without receiving and vice versa. UART0's registers are shown in Table e9.6.

To configure the UART, first set the baud rate. The UART uses the on-board TileLink bus clock, tlclk, as its clock source. For the FE310-G002, this bus clock is configured by default to be the same as the processor clock, coreclk, at 16 MHz. This clock signal must be divided down to produce the desired baud rate. The final baud rate is given by Equation 9.1:

$$f_{baud} = \frac{f_{in}}{div + 1} \tag{e9.1}$$

The FE310 UART peripheral supports only 8-N-1 and 8-N-2 protocol configurations. Both protocols support 8 data bits and no parity bit, and the packets can be configured to have either one stop bit (in 8-N-1) or two stop bits (in 8-N-2). The stop bit configuration is set in the `txctrl` register using the `nstop` field. By default, `nstop = 0`, which sets the peripheral to use one stop bit.

Data is transmitted and received using the `txdata` and `rxdata` registers, respectively. Both the transmit and receive registers are buffered by 8-entry, FIFO buffers. To transmit data, check that the full bit of the `txdata` register is 0, which indicates that there is room in the FIFO buffer for new data to be written. Then, write a byte to the data field in `txdata`. To read data, read the `rxdata` register and check that the empty bit is 0 to confirm that the byte in the data field is valid.

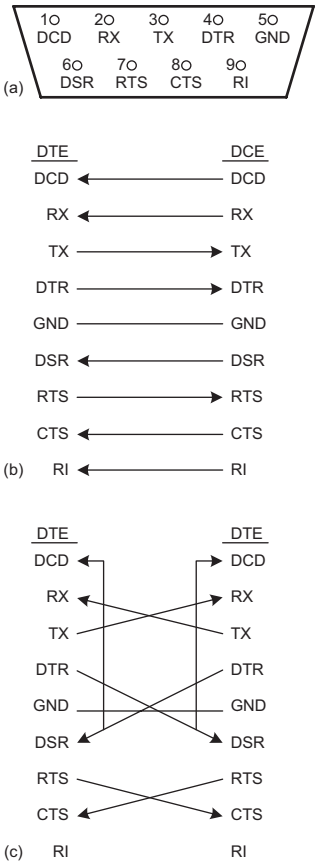


Figure e9.12 **DE-9 male cable (a) pinout, (b) standard wiring, and (c) null modem wiring**

Table e9.6 **UART memory mapped registers**

	...
0x10013018	div
	...
0x1001300C	rxctrl
0x10013008	txctrl
0x10013004	rxdata
0x10013000	txdata
	...

Adapted and printed with permission from Table 55 of the SiFive *FE310-G002 Manual*, © 2019 SiFive, Inc.

Example e9.4 SERIAL COMMUNICATION WITH A PC

Develop a circuit and a C program for an FE310 to communicate with a PC over a serial port at 115200 baud with 8 data bits, 1 stop bit, and no parity. The PC should be running a console program such as PuTTY[1] to read and write over the serial port. The program should ask the user to type a string. It should then indicate what the user typed.

Solution Figure e9.13(a) shows a basic schematic of the serial link illustrating the issues of level conversion and cabling. Because few PCs still have physical serial ports, we use a Plugable USB to RS-232 DB9 Serial Adapter from plugable.com shown in Figure e9.14 to provide a serial connection to the PC. The adapter connects to a female DE-9 connector soldered to wires that feed a transceiver, which

[1] PuTTY is available for free download at www.putty.org.

converts the voltages from the bipolar RS-232 levels to the FE310's 3.3 V level. The FE310 and PC are both DTE, so the TX and RX pins must be cross-connected in the circuit. The RTS/CTS handshaking from the FE310 is not used, and the RTS and CTS on the DE9 connector are tied together so that the PC will shake its own hand.

Figure e9.13(b) shows an easier approach with an Adafruit 954 USB to TTL serial cable. The cable is directly compatible with 3.3 V levels.

To configure PuTTY to work with the serial link, set *Connection type* to Serial and *Speed* to 115200. Set *Serial line* to the COM port assigned by the operating system to the Serial to USB Adapter. In

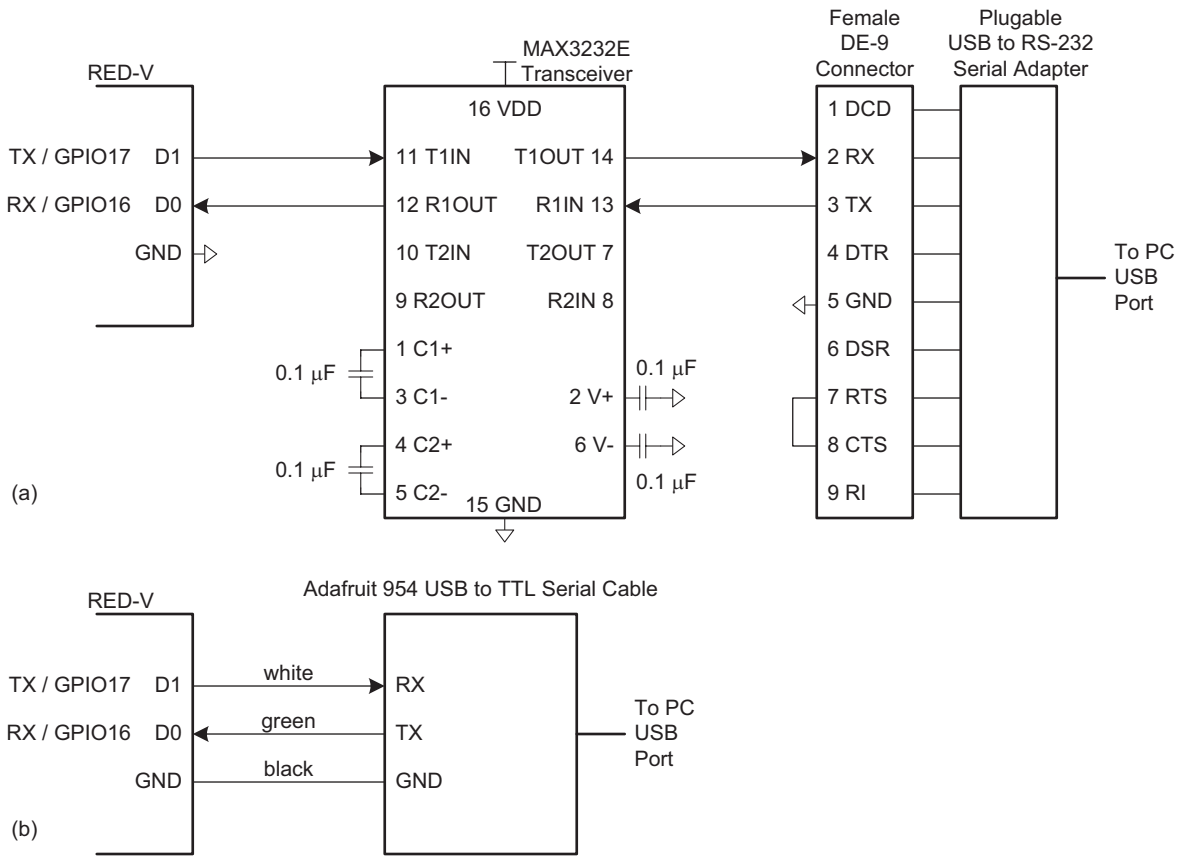


Figure e9.13 Serial communication link schematics: (a) serial communication via RS-232, (b) serial communication with USB to TTL serial cable

Windows, this can be found in the Device Manager; for example, it might be COM3. Under the *Connection → Serial* tab, set flow control to NONE or RTS/CTS. Under the Terminal tab, set Local Echo to Force On to have characters appear in the terminal as you type them.

The serial port device driver code in `EasyREDVIO.h` is shown in Code Example e9.5. The Enter key in the terminal program corresponds to a carriage return character represented as `\r` in C with an ASCII code of 0x0D. To advance to the beginning of the next line when printing, send both the `\n` and `\r` (new line and carriage return) characters.[2] The uartInit function configures the UART as described above. `getCharSerial` and `putCharSerial` read and write characters to the terminal, respectively, using the UART (Code Example e9.5).

Figure e9.14 Plugable USB to RS-232 DB9 serial adapter

Code Example e9.5 READING AND WRITING CHARACTERS (CHARS) TO A TERMINAL USING A UART

```
void uartInit(uint32_t baud) {
    uint32_t div = 16000000/baud-1;       // 16 MHz tileclock
    pinMode(16, GPIO_IOF0);
    pinMode(17, GPIO_IOF0);

    UART0->div.div = div;                 // Set clock divisor
    UART0->txctrl.txen = 1;               // Enable transmitter
    UART0->txctrl.nstop = 1;              // Set one stop bit
    UART0->rxctrl.rxen = 1;               // Enable receiver
}

uint8_t getCharSerial(void) {
    uart_rxdata_bits rxdata;              // Create temporary variable to store register

    while(1) {
        rxdata = UART0->rxdata;           // Read register exactly once
        if(!rxdata.empty) {
            return (uint8_t)rxdata.data; // Check to see if the data is valid
        }
    }
}

void putCharSerial(uint8_t c) {
    while(UART0->txdata.full);            // Wait until ready to transmit
    UART0->txdata.data = c;
}
```

The `main` function in Code Example e9.6 demonstrates printing to the console and reading from the console using the `putStrSerial` and `getStrSerial` functions.

[2] PuTTY prints correctly even if the `\r` is omitted.

Code Example e9.6 READING AND WRITING STRINGS TO A TERMINAL USING A UART

```
#include "EasyREDVIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
    int i = 0;
    do {                            // Read an entire string until detecting
        str[i] = getCharSerial(); // Carriage return
    } while ((str[i++] != '\r') && (i < MAX_STR_LEN));  // Look for carriage return
    str[i-1] = 0;                   // Null-terminate the string
}

void putStrSerial(char *str) {
    int i = 0;
    while (str[i] != 0) {           // Iterate over string
        putCharSerial(str[i++]); // Send each character
    }
}

int main(void) {
   char str[MAX_STR_LEN];

   uartInit(115200); // initialize UART with baud rate

   while(1) {
       putStrSerial("Please type something: \r\n");
       getStrSerial(str);
       putStrSerial("You typed: ");
       putStrSerial(str);
       putStrSerial("\r\n");
   }
}
```

Communicating with the serial port from a C program on a PC is a bit of a hassle because serial port driver libraries are not standardized across operating systems. Other programming environments such as Python, MATLAB, or LabVIEW make serial communication painless.

9.3.6 Timers

Embedded systems commonly need to measure time. For example, a microwave oven needs a timer to keep track of the time of day and another to measure how long to cook. It might use yet another to generate pulses to the motor spinning the platter and a fourth to control the power setting by only activating the microwave's energy for a fraction of every second.

The FE310 has a system timer with a 64-bit free-running counter that increments according to an externally provided clock signal. On the RED-V, this clock source is a 32.768 kHz oscillator (conveniently 2^{15} Hz). Figure e9.16 shows the memory map for the system timer. It is located within the core-local interruptor (CLINT) block. `mtime` contains the 64-bit current value of the counter. It can be read or written; so, to restart

the timer, a zero can be written. `mtimecmp` is a 64-bit register containing the timer comparison value and `msip` is the machine-mode software interrupt register. When the counter hits the value in `mtimecmp`, the least significant bit in the `msip` register is set to 1. Using the `msip` and `mtimecmp` registers is an efficient way to check that a delay has taken place. Table e9.7 shows the memory addresses for these registers.

If additional timers are needed, the PWM module (see Section 9.3.7.2) provides additional counters that can be used to measure precise delays.

Table e9.7 System timer registers

	...
0x0200BFFC	mtime (hi)
0x0200BFF8	mtime (lo)
	...
0x02004004	mtimecmp (hi)
0x02004000	mtimecmp (lo)
	...
0x02000000	msip
	...

Adapted and printed with permission from Table 24 of the SiFive *FE310-G002 Manual*, © 2019 SiFive, Inc.

Example e9.5 BLINKING LED

Write a program that blinks the status LED on the RED-V 5 times per second for 4 seconds.

Solution The delay function in EasyREDVIO (see Code Example e9.7) creates a delay of a specified number of milliseconds using the timer compare channel.

Code Example e9.7 DELAY FUNCTION

```
#define MTIME_CLK_FREQ 32768              // RTC frequency in Hz
volatile uint64_t *mtime = (uint32_t*) 0x0200BFF8;
void delay(int ms) {
    uint64_t doneTime = *mtime + (ms*MTIME_CLK_FREQ)/1000;
    while (*mtime < doneTime);            // Wait until time is reached
}
```

GPIO5 (D13) drives the activity LED on the RED-V board. The program in Code Example e9.8 sets this pin to be an output. It then turns the LED OFF and ON through a series of digital writes with a 200 ms repetition rate (5 Hz).

Code Example e9.8 BLINK ACTIVITY LED

```
#include "EasyREDVIO.h"

void main(void) {
    uint32_t i;

    pinMode(D13, OUTPUT);      // status led as output

    for(i = 0; i < 20; i++) {
        delay(100);
        digitalWrite(D13, 0); // turn led off
        delay(100);
        digitalWrite(D13, 1); // turn led on
    }
}
```

9.3.7 Analog I/O

The real world is an analog place. Many embedded systems need analog inputs and outputs to interface with the world. They use

analog-to-digital converters (ADCs) to quantize analog signals into digital values and digital-to-analog-converters (DACs) to do the reverse. Figure e9.15 shows symbols for these components. Such converters are characterized by their resolution, dynamic range, sampling rate, and accuracy. For example, an ADC might have $N = 12$-bit resolution over a range V_{ref}^- to V_{ref}^+ of 0 to 5 V with a sampling rate of $f_s = 44$ kHz and an accuracy of ±3 least significant bits (lsbs). Sampling rates are also listed as samples per second (sps), where 1 sps = 1 Hz. The relationship between the analog input voltage $V_{\text{in}}(t)$ and the digital sample $X[n = t / f_s]$ is

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref}^-}{V_{ref}^+ - V_{ref}^-} \tag{e9.2}$$

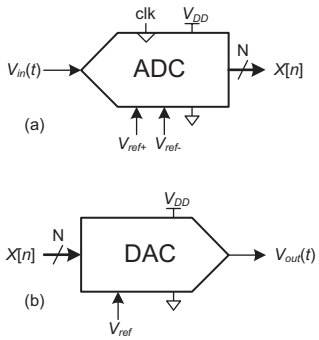


Figure e9.15 ADC and DAC symbols

For example, an input voltage of 2.5 V (half of full scale) would correspond to $2^{12}/2$ (half of the maximum value), that is, an output of $100000000000_2 = 0x800 = 2^{11} = 2048$, with an uncertainty of up to 3 lsbs.

Similarly, a DAC might have $N = 16$-bit resolution over a full-scale output range of $V_{ref} = 2.56$ V. It produces an output of

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref} \tag{e9.3}$$

Many microcontrollers have built-in ADCs of moderate performance. For higher performance (e.g., 16-bit resolution or sampling rates in excess of 1 MHz), it is often necessary to use a separate ADC connected to the microcontroller. Fewer microcontrollers have built-in DACs, so separate chips must be used to convert digital values to an analog voltage. However, microcontrollers often produce analog outputs using a technique called pulse-width modulation (PWM).

D/A Conversion

The FE310 has no general-purpose DAC. This section describes D/A conversion using external DACs and illustrates interfacing the FE310 with other chips over parallel and serial ports. The next section achieves the same result using PWM.

Some DACs accept the N-bit digital input on a parallel interface with N wires, while others accept it over a serial interface, such as SPI. Some DACs require both positive and negative power supply voltages, while others operate off of a single supply. Some support a flexible range of supply voltages, while others demand a specific voltage. The input logic levels should be compatible with the digital source. Some DACs

produce a voltage output proportional to the digital input, while others produce a current output; an operational amplifier may be needed to convert this current to a voltage in the desired range.

In this section, we use the Linear Technology LTC1450 12-bit parallel DAC and the Microchip MCP4801 8-bit serial DAC. Both produce voltage outputs, run off a single 5 to 15 V power supply, use $V_{IH} = 2.4\,V$ such that they are compatible with 3.3 V I/O, come in DIP packages that make them easy to breadboard, and are easy to use. The LTC1450 produces an output on a scale of 0 to 2.048 V or 0 to 4.095 V depending on the gain setting, consumes 2 mW, comes in a 24-pin package, and has a 4 µs settling time, permitting an output rate of 250 ksamples/second. The datasheet is at analog.com. The MCP4801 produces an output on a scale of 0 to 2.048 V or 0 to 4.096 V, consumes less than 2 mW, comes in an 8-pin package, and has a 4.5 µs settling time. Its SPI operates at a maximum of 20 MHz. The datasheet is at microchip.com.

Example e9.6 ANALOG OUTPUT WITH EXTERNAL DACS

Sketch a circuit and write the software for a simple signal generator producing sine and triangle waves using a RED-V, an LTC1450, and an MCP4801.

Solution The circuit is shown in Figure e9.16. Two DAC chips are used in this example. Both DACs use a 5 V power supply and have a 0.1 µF decoupling capacitor to reduce power supply noise.

The LTC1450 DAC has 12 data inputs, D0 to D11, that specify the analog voltage to generate on VOUT. In our example, we use only 8-bit precision, so we tie the four least significant bits, D0 to D3, to ground. To load data into the DAC, the RED-V puts the desired value on D4 to D11. Then, the RED-V drives the active-low write (WR) pin low to write the data to the DAC. CLR is tied to VCC because we don't need to clear the input data latches. LDAC, the low-asserted load DAC signal, is tied to GND to load data every time WR goes low.

The MCP4801 connects to the RED-V via SPI1. In addition to the standard SPI signals, the MCP4801 has an analog output voltage pin (VOUT) and two active-low control inputs: hardware shutdown (SHDN) and latch DAC (LDAC). SHDN is used to turn off the output driving circuitry and save power when the output value is not needed. The LDAC latches the input values when it is low. To send data to the MCP4801, a 16-bit value is sent over SPI: bits 11 to 4 hold D7 to D0; bit 13 is the gain selection (1x if set to 1, 2x if set to 0); and bit 12 controls SHDN (0 shuts down the output, 1 allows VOUT to drive an analog value). In this case, SHDN is controlled in software, so it is left floating (not driven) in the circuit.

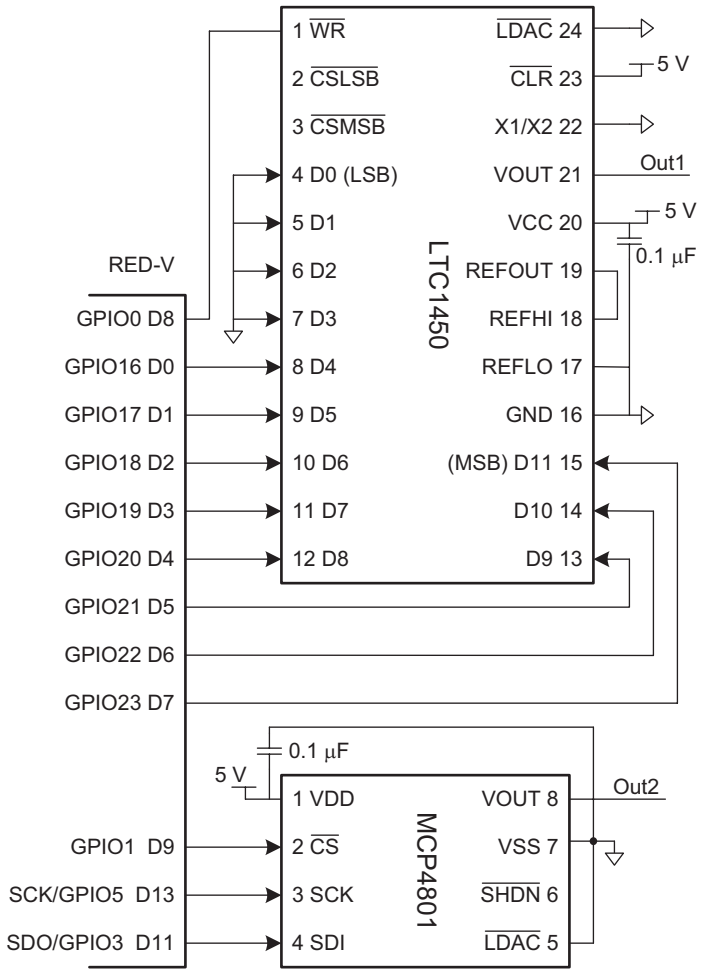


Figure e9.16 DAC parallel and serial interfaces to a RED-V board

The program for driving both DACs is shown in Code Example e9.9. The program configures the 8 parallel port pins as outputs and also configures GPIO0 as an output to drive the `WR` signal on the LTC1450 and GPIO1 to drive the chip select signal on the MCP4801. It initializes the SPI to 500 kHz. `initWaveTables` precomputes an array of sample values for the sine and triangle waves. It then updates the serial DAC. Then, the program delays until the timer indicates that it is time for the next sample. The maximum frequency of the generated waveforms is set by the time to send each point in the `genWaves` function, which is limited by the SPI transmission time.

Code Example e9.9 GENERATING A SINE WAVE USING A DAC

```
#include "EasyREDVIO.h"
#include <math.h> // required to use the sine function

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];

#define SHDNn_Pos 12
#define Gain_Pos  13

int parallelPins[8] = {D0, D1, D2, D3, D4, D5, D6, D7};

void initWaveTables(void) {
  int i;
  for (i = 0; i<NUMPTS; i++) {
    sine[i] = 127*(sin(2*3.14159*i/NUMPTS) + 1); // 8-bit scale
    if (i < NUMPTS/2) triangle[i] = i*255/NUMPTS; // 8-bit scale
    else triangle[i] = 254 - i*255/NUMPTS;
  }
}

void genWaves(int freq) {
  int i, j;
  int delay_cycles = MTIME_CLK_FREQ/(NUMPTS*freq);

  for (i = 0; i < 2000; i++){
    for (j = 0; j < NUMPTS; j++) {
      uint64_t doneTime = *mtime + delay_cycles; // Set sample period

      // Load serial DAC
      digitalWrite(1, 0);       // enable chip (chip select: CS = 0)
      // Set SHDNn to active (bit 12) and gain to 1 (bit 13)
      volatile uint16_t sine_samp_dac = ((uint16_t) sine[j] << 4) \
                                        |(1 << SHDNn_Pos) | (1 << Gain_Pos);
      spiSendReceive16(sine_samp_dac);
      digitalWrite(1, 1);       // disable chip (chip select: CS = 1)

      // Load parallel DAC
      digitalWrite(0, 1);       // No load while changing inputs
      digitalWrites(parallelPins, 8, triangle[j]);
      digitalWrite(0, 0);       // Load new points into DACs
      while(*mtime < doneTime); // Wait for mtime_cmp to hit
    }
  }
}

int main(void) {
  pinsMode(parallelPins, 8, OUTPUT); // Set pins connected to the AD558 as outputs
  pinMode(0, OUTPUT);                // Make pin 0 an output to control LOAD
  pinMode(1, OUTPUT);                // Make pin 1 an output to control CE
  spiInit(15, 0, 0);                 // Initialize the SPI
  initWaveTables();
  genWaves(100);
}
```

Pulse-Width Modulation

Another way for a digital system to generate an analog output is with *pulse-width modulation* (PWM), in which a periodic output is pulsed high for part of the period and low for the remainder. The duty cycle is the fraction of the period for which the pulse is high (pulse width/period), as shown in Figure e9.17. The average value of the output is

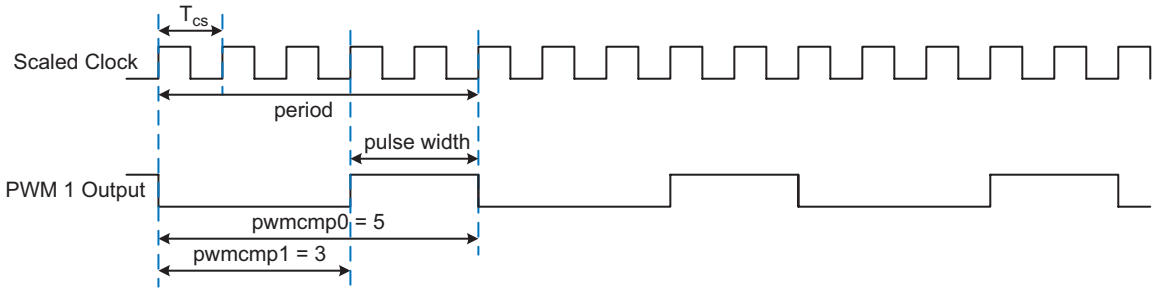


Figure e9.17 **Pulse-width modulated (PWM) signal**

proportional to the duty cycle. For example, if the output swings between 0 and 3.3 V and has a duty cycle of 25%, the average value will be $0.25 \times 3.3 = 0.825$ V. Low-pass filtering a PWM signal eliminates the oscillation and leaves a signal with the desired average value. Thus, PWM is an effective way to produce an analog output if the pulse rate is much higher than the analog output frequencies of interest. Other applications of PWM include making square wave audio tones and digital control of a motor or light at partial power or brightness.

The FE310 has three PWM peripherals and, as shown in Table e9.3, each PWM has four PWM outputs, for a total of 12 available PWM outputs. The outputs on PWM0 have 8-bit precision, and PWM1 and PWM2 have 16-bit precision. In this section, we show how to use PWM2, but configuring and using the other two PWM peripherals follows similar steps. PWM2 has four outputs (PWM2_PWM0, PWM2_PWM1, PWM2_PWM2, PWM2_PWM3) that are available on pins GPIO10-13 using pin function IOF1.

PWMs have several waveform generation modes, but we focus on generating PWM waveforms such as those in Figure e9.17. To do this, the peripheral is configured in a repeating mode in which comparator 0 (`pwmcmp0`) sets the period and comparator 1 (`pwmcmp1`) sets the low time. These times are in units of a scaled clock period, T_{cs}. For example, as shown in Figure e9.17, if the scaled clock period is 0.5 µs (2 MHz) and `pwmcmp0` = 5, then PWM2_PWM1 (pin 11) will oscillate at a period of 5×0.5 µs = 2.5 µs (400 kHz). If `pwmcmp1` = 3, then the duty cycle is $1-(3/5) = 40\%$.

Table e9.8 shows the memory map for the PWM2 registers. In this section, we describe the steps needed to configure the PWM1_PWM1 output; the other PWMs and their outputs are configured using a similar procedure.

Table e9.9 shows the bitfields in the PWM configuration register `pwmcfg`. Note that most bits are not cleared on system reset,

Table e9.8 **PWM2 configuration registers**

0x1002502C	pwmcmp3
0x10025028	pwmcmp2
0x10025024	pwmcmp1
0x10025020	pwmcmp0
...	...
0x10025010	pwms
0x1002500C	...
0x10025008	pwmcount
0x10025004	...
0x10025000	pwmcfg
	...

Adapted and printed with permission from Table 89 of the SiFive *FE310-G002 Manual*, © 2019 SiFive, Inc.

Table e9.9 PWM configuration register fields

PWM Configuration Register (pwmcfg)				
Register Offset		0x0		
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[3:0]	pwmscale	RW	X	PWM Counter scale
[7:4]	Reserved			
8	pwmsticky	RW	X	PWM Sticky - disallow clearing pwmcmp*X*ip bits
9	pwmzerocmp	RW	X	PWM Zero - counter resets to zero after match
10	pwmdeglitch	RW	X	PWM Deglitch - latch pwmcmp*X*ip within same cycle
11	Reserved			
12	pwmenalways	RW	0x0	PWM enable always - run continuously
13	pwmenoneshot	RW	0x0	PWM enable one shot - run one cycle
[15:14]	Reserved			
16	pwmcmp0center	RW	X	PWM0 Compare Center
17	pwmcmp1center	RW	X	PWM1 Compare Center
18	pwmcmp2center	RW	X	PWM2 Compare Center
19	pwmcmp3center	RW	X	PWM3 Compare Center
[23:20]	Reserved			
24	pwmcmp0gang	RW	X	PWM0/PWM1 Compare Gang
25	pwmcmp1gang	RW	X	PWM1/PWM2 Compare Gang
26	pwmcmp2gang	RW	X	PWM2/PWM3 Compare Gang
27	pwmcmp3gang	RW	X	PWM3/PWM0 Compare Gang
28	pwmcmp0ip	RW	X	PWM0 Interrupt Pending
29	pwmcmp1ip	RW	X	PWM1 Interrupt Pending
30	pwmcmp2ip	RW	X	PWM2 Interrupt Pending
31	pwmcmp3ip	RW	X	PWM3 Interrupt Pending

so it is prudent to start by resetting all bits to 0, then writing a 1 to `pwmenalways` and `pwmzerocmp` to configure the PWM to generate a repeating waveform with the period set by `pwmcmp0`.

The scaled clock frequency f_{scaled} is the base bus clock frequency of f_{base} = 16 MHz divided by $2^{\texttt{pwmscale}}$, where `pwmscale` is a 4-bit number in the range of 0 to 15 in the `pwmcfg` register. The PWM frequency is $f_{pwm} = f_{scaled}/\texttt{pwmcmp0} = f_{base}/(\texttt{pwmcmp0} \times 2^{\texttt{pwmscale}})$. As described above, the duty cycle is 1−(`pwmcmp1/pwmcmp0`). Many possible choices exist for `pwmscale` and `pwmcmp0` to give a desired PMW frequency. However, the PWM frequency resolution (error between desired and actual frequency) is best when `pwmscale` is as small as possible and `pwmcmp0` is as large as possible, within the constraint that `pwmcmp0` is an unsigned 16-bit number (i.e., it cannot exceed 65,535).

Example e9.7 PULSE-WIDTH MODULATION (PWM)

Choose `pwmscale` and `pwmcmp0` to blink an LED at 1.2 Hz. Repeat the question to generate a tone of 1190 Hz.

Solution To illustrate this, suppose that we wished to blink an LED at f_{pwm} = 1.2 Hz. $f_{pwm} = f_{base}/(\texttt{pwmcmp0} \times 2^{\texttt{pwmscale}})$. Thus, choose `pwmscale` = 8 and `pwmcmp0` = 52083.33 to get the desired frequency with f_{scaled} = 16 MHz/2^8 = 62.5 KHz. `pwmcmp0` is a 16-bit register, so we must round to 52083, giving an actual f_{pwm} = 16 MHz/(52083 × 2^8) = 1.20000768 Hz, which is very close to the desired frequency and comparable to the 10 parts per million accuracy of a typical quartz crystal clock reference.

On the other hand, suppose that we wanted a 1190 Hz output. If we didn't change `pwmscale`, `pwmcmp0` would need to be 52.521. Rounding to 53 gives an actual f_{pwm} = 16 MHz/(53 × 2^8) = 1179.2 Hz, an error of about 10 Hz, or 0.91%. If we needed a more accurate output frequency, we could reduce `pwmscale` to 0 and increase `pwmcmp0` to 13445, obtaining f_{pwm} = 16 MHz/(13445 × 2^0) = 1190.03 Hz.

A PWM device driver could have `pwmInit()` and `pwm(int freq, float duty)` functions. `pwmInit` would set the appropriate pin for the PWM peripheral and set the bits in the `pwmcfg` register. The `pwm` function would choose the appropriate `pwmscale`, `pwmcmp0`, and `pwmcmp1` to generate a waveform with the specified frequency and duty cycle. Writing these functions is similar to writing the SPI or UART device driver; details are left as an exercise for the reader.

A/D Conversion

Many microcontrollers have at least one built-in ADC, but the FE310 does not. This section describes A/D conversion using an external converter similar to the external DACs described in the prior section.

Example e9.8 ANALOG INPUT WITH AN EXTERNAL ADC

Interface a 10-bit MCP3002 A/D converter to an FE310 using SPI and print the input value. Set a full-scale voltage of 3.3 V. Search for the datasheet on the web for full details of operation.

Solution Figure e9.18 shows a schematic of the connection between the FE310 and the MCP3002 ADC and Code Example e9.10 shows the driver code. The MCP3002 uses VDD as its full-scale reference: that is, VDD (Pin 8) is connected to 3.3 V. It can accept a 3.3 to 5.5 V supply; we choose 3.3 V. The ADC

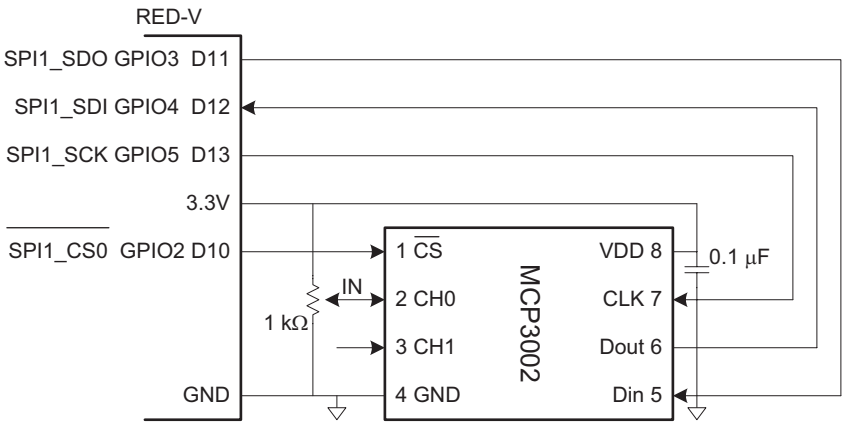


Figure e9.18 Reading an analog input using an external ADC

has two input channels, CH0 and CH1. We connect channel 0 to a potentiometer (not shown in the figure) that we rotate to adjust the input voltage between 0 and 3.3 V.

The FE310 code (see Code Example e9.10) initializes the SPI and repeatedly reads and prints samples. According to the datasheet, the FE310 must send the 16-bit quantity 0x6000 over SPI to read CH0 and will receive the 10-bit result back in the bottom 10 bits of the 16-bit result. Since we cannot directly set the FE310 to transmit 16-bit frames, we can put together two 8-bit packets without raising the chip select line in between. Although the SPI peripheral has the option to control the chip select line automatically, here we manually configure GPIO2 as an output and toggle it appropriately at the beginning of the transmission (writing the chip select line to 0) and the end of the transmission (writing the chip select line to 1).

Code Example e9.10 CODE FOR INTERFACING WITH ADC

```
#include "EasyREDVIO.h"

int main(void) {
  uint8_t sample;
  spiInit(15, 0, 0);   // Initialize the SPI
                       // Clock divisor of div = 15, CPOL = 0, CPHA = 0
  pinMode(D10,  OUTPUT);
  while(1) {
    digitalWrite(D10, 0);
    spiSendReceive('0x60');
    sample = spiSendReceive('0x00');
    digitalWrite(D10, 1);
    printf("Read %d\n", sample);
    delay(200);
  }
}
```

9.3.8 Interrupts

So far, we have relied on *polling*, in which the program continually checks a value until an event occurs such as data arriving on a UART or a timer reaching its compare value. This can be a waste of the processor's power and makes it difficult to write programs that do interesting work while simultaneously waiting for events to occur.

Most microcontrollers support *interrupts*. When an event occurs, the microcontroller stops the executing program and jumps to an interrupt handler that responds to the interrupt. After handling the interrupt, the processor then returns to the user program and seamlessly continues where it was interrupted. Interrupts are the hardware exceptions discussed in Section 6.6.2.

The FE310 has a core-local interruptor (CLINT) that handles timer and software interrupts. Software interrupts are used for interprocessor communication and debugging. The FE310 also has a platform-level interrupt controller (PLIC) that collects interrupts from other peripherals. In a multiprocessor system, the PLIC routes the peripheral interrupt to an appropriate processor to handle it.

Example e9.9 shows how to blink an LED using interrupts instead of polling.

Example e9.9 BLINKING AN LED WITH A TIMER INTERRUPT

We configure local interrupts on the FE310 using the CLINT. For the FE310-G002 chip used on the RED-V RedBoard and RED-V Thing Plus, information on how to use interrupts is provided in Chapters 8 to 10 of the *FE310-G002 Manual*. The basic configuration procedure for local interrupts through the CLINT is outlined below.

1. Write a trap handler to handle execution whenever an interrupt or exception is triggered. The main purpose of the trap handler is to figure out what interrupt or exception was triggered and then to perform the desired operation in response.
2. Configure `mtvec`, a control and status register (CSR), with the address of the trap handler and the mode (direct or vectored).
3. Enable the specific interrupt (e.g., from the timer)
4. Globally enable all interrupts.

After defining constants and function pointer arrays, the code declares the global trap handler function `handle_trap()`, as shown in Code Example e9.11. This function is called whenever we trigger a trap (interrupt or exception). Its job is to figure out which trap triggered the call and jump to the correct interrupt or exception handler. The trap handler performs two tasks. First, it uses a mask (`MCAUSE_INT_MASK`) to check the most significant bit of the `mcause` register,

which indicates whether the trap is an interrupt (generated from a device external to the core) or an exception (generated internally in the core). The structure of `mcause` is shown in Table e9.10 and the listing of interrupt and exception codes is shown in Table 6.6. Then, it uses an additional mask (`MCAUSE_CODE_MASK`) to determine the trap code and jumps to the appropriate interrupt or exception handler based on the index of the `interrupt_handler` or `exception_handler` function pointer arrays.

Code Example e9.11 SETTING UP THE TRAP HANDLER

```
// Function pointer arrays for interrupt and exception handlers
#define MAX_INTERRUPTS 16
void (*interrupt_handler[MAX_INTERRUPTS])();
void (*exception_handler[MAX_INTERRUPTS])();

// Masks for isolating interrupt vs. exception and the relevant code
#define MCAUSE_INT_MASK 0x80000000  // If [31] = 1 interrupt, else exception
#define MCAUSE_CODE_MASK 0x7FFFFFFF // low bits show code

// Declaration for interrupt handler. Declared with attribute interrupt which
// maps to GCC helper function.
void handle_trap(void) __attribute((interrupt));

// Define trap handler
void handle_trap() {
    unsigned long mcause_value = read_csr(mcause);
    if (mcause_value & MCAUSE_INT_MASK) {
        // Branch to interrupt handler here
        // Index into 32-bit array containing addresses of functions
        interrupt_handler[mcause_value & MCAUSE_CODE_MASK]();
    }
    else {
        // Branch to exception handler here
        exception_handler[mcause_value & MCAUSE_CODE_MASK]();
    }
}
```

Next, we define an *interrupt service routine* (*ISR*) for the timer. This is a function that contains instructions we want to execute whenever we get a timer interrupt. In this example, we call this function `timer_handler()`. It reads the current value of the GPIO pin driving the on-board LED (D13/GPIO5) and toggles the state using `digitalWrite()`.

Table e9.10 `mcause` register fields

Bits	Field Name	Description
[9:0]	Exception Code	A code identifying the most recent exception
[30:10]	Reserved	
31	Interrupt	1 if trap was caused by an interrupt; 0 otherwise

Then, it resets the timer by calling `reset_timer()`, which sets the current count in the `mtime` register to 0 and resets the count value at which the next interrupt should be triggered.

Code Example e9.12 TIMER ISR AND FUNCTION TO RESET TIMER

```
void timer_handler() {
    volatile int pin_val = (GPIOO->output_val >> D13) & 1; // Read the current output state
    if(pin_val) digitalWrite(D13, LOW);
    else digitalWrite(D13, HIGH);
    reset_timer(MTIME_CLK_FREQ / (2 * BLINK_FREQ));
}

void reset_timer(int count_val) {
    *MTIME = 0;
    *MTIMECMP = count_val;
}
```

Unlike the other registers we have used in this chapter, most of the registers related to the CLINT—such as `mtvec`, `mie`, and `mstatus`—are not memory mapped. These registers are called *control and status registers* (*CSRs*). To manipulate CSRs, we must use the RISC-V assembly instructions CSR read (`csrr`) and CSR write (`csrw`). These instructions can conveniently be wrapped in C macros to enable us to more easily interact with them.

Code Example e9.13 MACROS FOR WRITING AND READING CSRs

```
// Macros for reading and writing the control and status registers (CSRs)
#define read_csr(reg) ({ unsigned long __tmp; \
 asm volatile ("csrr %0, " #reg : " = r"(__tmp)); \
 __tmp; })

#define write_csr(reg, val) ({ \
 asm volatile ("csrw " #reg ", %0" :: "rK"(val)); })
```

After setting up the trap handler, we register it by placing its address in the `mtvec` register. Its structure is shown in Table e9.11. `mtvec` is a 32-bit register where bits [31:2] hold bits [31:2] of the address of the trap handler function (bits [1:0] are automatically assumed to be 0 since the instructions must be word aligned in the memory). Bits [1:0] of `mtvec` are instead used to configure whether the exceptions are handled in direct or vector mode. In direct mode, regardless of what interrupt or exception fires, we jump to the function address indicated by `mtvec[31:2]`. This is the mode we will use here. In vectored mode, we jump to different memory addresses depending on what interrupt is triggered.

After configuring the trap handler and setting up the timer interrupt service routine, we finish by enabling the machine timer interrupt by setting bit 7, the machine timer interrupt enable (`MTIE`) bit, in the machine interrupt enable `mie` register and by enabling interrupts globally by

Table e9.11 `mtvec` **register fields**

Bits	Field Name	Description
[1:0]	MODE	Sets the interrupt processing mode to direct (`00`) or vectored (`10`)
[31:2]	BASE[31:2]	Base address of the `trap_handler`

Adapted and printed with permission from Table 18 of the SiFive *FE310-G0002 Manual*, © 2019 SiFive, Inc.

Code Example e9.14 FUNCTIONS TO REGISTER TRAP HANDLER BY WRITING TO `mtvec`

```
void register_trap_handler(void *func) {
    // Set mtvec[31:2] to interrupt handler function address
    // The two lsbs are not meaningful because instructions are aligned to 4 bytes
    // Set mtvec[1:0] to 00 for direct mode.
    write_csr(mtvec, ((unsigned long) func) & ~(0b11));
}
```

setting bit 3, the machine interrupt enable (`MIE`) bit, in the machine status register (`mstatus`). Simple helper functions for globally enabling and disabling interrupts are shown in Code Example e9.15. Complete details about the structure of `mstatus` and `mie` can be found in Tables 17 and 20 in the SiFive *FE310-G002 Manual*.

Code Example e9.15 GLOBALLY ENABLE OR DISABLE INTERRUPTS

```
void enable_interrupts() {
    // Set bit 3 in mstatus (MIE) to enable machine interrupts
    write_csr(mstatus, read_csr(mstatus) | (1 << 3));

void disable_interrupts() {
    // Clear bit 3 in mstatus (MIE) to disable machine interrupts
    write_csr(mstatus, read_csr(mstatus) & ~(1 << 3));
}
```

Finally, we put all the pieces together and call the functions we built in our main function, as shown in Code Example e9.16. Here, because our application is interrupt driven, we don't do anything in the main `while` loop.

Care should be taken when developing safety- or timing-critical applications with interrupts as they are asynchronous events and can be triggered at any time during program execution. You as a programmer should consider what bugs may be introduced by an interrupt triggering at an inopportune time. If you have a segment of code where you want to avoid being interrupted, you can disable interrupts (i.e., clear `MIE` in

Code Example e9.16 BLINK LED WITH TIMER INTERRUPTS

```
#include "EasyREDVIO.h"

// CLINT memory map pointers
#define MTIMECMP ((uint64_t *) 0x02004000UL)
#define MTIME ((uint64_t *) 0x0200BFF8UL)

#define BLINK_FREQ 4 // This is an arbitrary constant used to specify the LED blink
frequency

int main(void) {

    // Set LED pin as an output
    pinMode(D13, OUTPUT);

    // Register interrupt handler.
    // The machine timer interrupt is exception code 7 as shown in  so we put the
    // timer_handler() function at index 7 of the array.
    interrupt_handler[7] = timer_handler;

    // Set up interrupt by configuring mtvec
    register_trap_handler(handle_trap);

    // Reset timer
    reset_timer(MTIME_CLK_FREQ / (2 * BLINK_FREQ));

    // Enable timer interrupt
    write_csr(mie, read_csr(mie) | (1 << 7));

    enable_interrupts();

    while(1) {
    };

    return 0;
}
```

`mstatus`) when executing the instructions and then re-enable interrupts when finished (i.e., set `MIE` in `mstatus`).

9.4 OTHER MICROCONTROLLER PERIPHERALS

Microcontrollers frequently interface with other external peripherals. This section describes a variety of common examples, including character-mode liquid crystal displays (LCDs), VGA monitors, Bluetooth wireless links, and motor controllers.

9.4.1 Character LCDs

A character LCD is a small liquid crystal display capable of showing one or a few lines of text. They are commonly used in the front panels of appliances such as cash registers, laser printers, and fax machines that need to display a limited amount of information. They are easy to interface with a microcontroller over parallel, RS-232, or SPI interfaces. Crystalfontz America sells a wide variety of character LCDs ranging from 8 columns × 1 row to 40 columns × 4 rows with choices of color, backlight, 3.3 or 5 V

operation, and daylight visibility. Their LCDs can cost $20 or more in small quantities, but prices come down to under $5 in high volume.

This section shows how to interface the RED-V board to the Crystalfontz CFAH2002A-TMI-JT 20 × 2 parallel LCD shown in Figure e9.19. The interface is an 8-bit parallel interface, which is compatible with the industry-standard HD44780 LCD controller originally developed by Hitachi.

Figure e9.20 shows the LCD connected to a RED-V board over an 8-bit parallel interface (inputs D0-D7 on the LCD). The LCD logic operates at 5 V but is compatible with 3.3 V inputs from the RED-V board. The LCD contrast is set by a second voltage (input to pin 3, VO) produced using a potentiometer; it is usually most readable at a setting of 4.2 to 4.8 V. The LCD receives three control signals: RS (1 for characters, 0 for instructions), R/$\overline{W}$ (1 to read from the display, 0 to write), and E (pulsed high for at least 250 ns to enable the LCD when the next data byte is ready to be written to it). In addition to sending data bits, the data lines (D0–D7) are used to set LCD configurations when RS = 0 (i.e., when in instruction mode). When read, LCD port D7 returns the busy flag, which is 1 when the LCD is busy and 0 when it is ready to accept another instruction or byte of data.

To initialize the LCD, the RED-V board must write a sequence of instructions to the LCD as given in Table e9.12. Instructions are written by making RS = 0 and R/$\overline{W}$ = 0, putting the value on the eight data lines, and pulsing E for at least 250 ns. Data bytes are written by doing the same thing except making RS = 1. After sending an instruction or data byte, the processor must wait for at least a specified amount of time (or sometimes until the busy flag is clear) before sending another instruction or data byte. The busy flag (D7) is read by making RS = 0 and R/$\overline{W}$ = 1 and pulsing E for at least 250 ns. Remember that GPIO23 must also be temporarily set as an input when reading the busy flag (D7).

After configuration is complete, the LCD is ready to accept text to display. Write text to the LCD by making RS = 1 and R/$\overline{W}$ = 0, putting

Figure e9.19 Crystalfontz CFAH2002A-TMI 20 × 2 character LCD

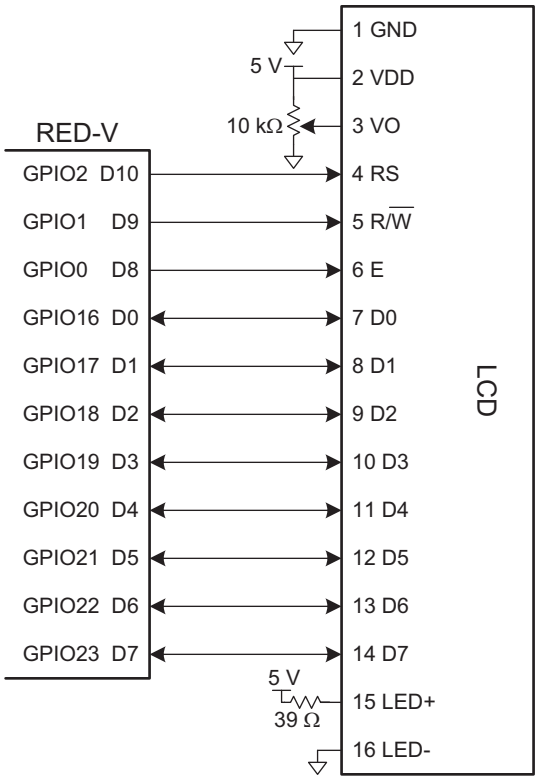


Figure e9.20 **Parallel LCD interface**

the value on the eight data lines, and pulsing E for at least 250 ns. After each character, the RED-V must wait for the busy bit to clear before sending another character. It may also send the instruction 0x01 to clear the display or 0x02 to return to the home position in the upper left.

Example e9.10 LCD CONTROL

Write a program to print "I love LCDs" to the Crystalfontz CFAH2002A-TMI character display.

Solution The program in Code Example e9.17 writes "I love LCDs" to the display by initializing the display and then sending the characters.

9.4.2 VGA Monitor

A more flexible display option is to drive a computer monitor. This section explains the low-level details of driving a VGA (*video graphics array*) monitor directly from an FPGA.

Code Example e9.17 WRITING "I LOVE LCDS" TO LCD

```
#include "EasyREDVIO.h"

int LCD_IO_Pins[] = {D0, D1, D2, D3, D4, D5, D6, D7};

typedef enum {INSTR, DATA} mode;
#define RS D10
#define RW D9
#define E  D8

char lcdRead(mode md) {
    char c;
    pinsMode(LCD_IO_Pins, 8, INPUT);
    digitalWrite(RS,(md == DATA));      // set instr/data mode
    digitalWrite(RW, 1);                // RWbar = read mode
    digitalWrite(E, 1);                 // pulse enable
    delay(1);                           // wait for LCD response
    c = digitalReads(LCD_IO_Pins, 8);   // read a byte from parallel port
    digitalWrite(E, 0);                 // turn off enable
    delay(1);
    return c;
}

void lcdBusyWait(void) {
    char state;
    do {
        state = lcdRead(INSTR);
    } while(state & 0x80);
}

void lcdWrite(char val, mode md) {
    pinsMode(LCD_IO_Pins, 8, OUTPUT);
    digitalWrite(RS, (md == DATA));     // set instr/data mode. OUTPUT = 1, INPUT = 0
    digitalWrite(RW, 0);                // set RW pin to write  (RW = 0)
    digitalWrites(LCD_IO_Pins, 8, val); // write the char to the parallel port
    digitalWrite(E, 1); delay(1);       // pulse E
    digitalWrite(E, 0); delay(1);
}

void lcdClear(void) {
    lcdWrite(0x01, INSTR); delay(1);
}

void lcdPrintString(char* str) {
    while (*str != 0) {
        lcdWrite(*str, DATA); lcdBusyWait();
        str++;
    }
}

void lcdInit(void) {
    pinMode(RS, OUTPUT); pinMode(RW, OUTPUT); pinMode(E,OUTPUT);
    // send initialization routine:
    delay(15);
    lcdWrite(0x30, INSTR); delay(1);
    lcdWrite(0x30, INSTR); delay(1);
    lcdWrite(0x30, INSTR); lcdBusyWait();
    lcdWrite(0x3C, INSTR); lcdBusyWait();
    lcdWrite(0x08, INSTR); lcdBusyWait();
    lcdClear();
    lcdWrite(0x06, INSTR); lcdBusyWait();
    lcdWrite(0x0C, INSTR); lcdBusyWait();
}

int main(void) {
    lcdInit();
    lcdPrintString("I love LCDs!");
}
```

Table e9.12 LCD initialization sequence

Code (D7-D0)	Purpose	Wait (μs)
(apply VDD)	Allow device to turn on	15000
0x30	Set 8-bit mode	4100
0x30	Set 8-bit mode again	100
0x30	Set 8-bit mode yet again	Until busy flag is clear
0x3C	Configure 2 lines and 5 × 8 dot font	Until busy flag is clear
0x08	Turn display OFF	Until busy flag is clear
0x01	Clear display	1530
0x06	Set entry mode that increments cursor after each character	Until busy flag is clear
0x0C	Turn display ON with no cursor	

(These are instructions: so RS = 0 and $\mathbf{R}/\overline{W} = 0$.)

The VGA monitor standard was introduced in 1987 for the IBM PS/2 computers, with a 640 × 480 pixel resolution on a *cathode ray tube* (CRT) and a 15-pin connector conveying color information with analog voltages. Modern LCD monitors have higher resolution but remain backward compatible with the VGA standard.

In a CRT, an electron gun scans across the screen from left to right, exciting fluorescent material to display an image. Color CRTs use three different phosphors for red, green, and blue, and three electron beams. The strength of each beam determines the intensity of each color in the pixel. At the end of each scan line, the gun must turn off for a *horizontal blanking interval* to return to the beginning of the next line. After all of the scan lines are complete, the gun must turn off again for a *vertical blanking interval* to return to the upper left corner. This entire process repeats about 60 to 75 times per second to refresh the fluorescence and give the visual illusion of a steady image. Modern displays typically use LCD technology, which doesn't require the same electron scan gun but uses the same VGA interface timing for compatibility.

In a 640 × 480 pixel VGA monitor, the full frame is actually 800 pixels × 525 horizontal scan lines as shown in Figure e9.21, but only 480 of the scan lines and 640 pixels per scan line actually convey the image, while the remainder are black. A scan line begins with a 48-pixel *back porch*, the blank section on the left edge of the screen. It then contains 640 active pixels, followed by a blank 16-pixel *front porch* at the right edge of the screen and a 96-pixel clock horizontal sync (hsync) pulse to rapidly move

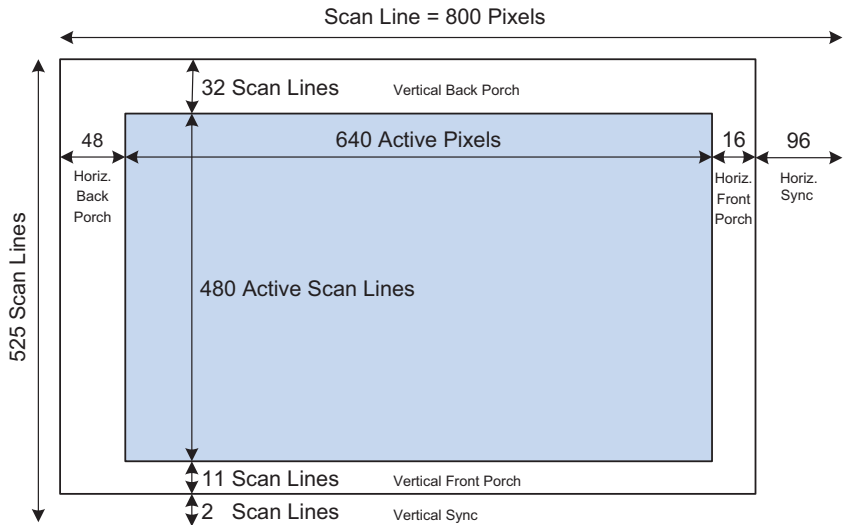


Figure e9.21 VGA frame

the gun back to the left edge. In the vertical direction, the screen starts with a 32-scan line back porch at the top, followed by 480 active scan lines, a front porch of 11 scan lines at the bottom and a 2-scan line vertical sync (vsync) pulse to return to the top to start the next frame. For a 640×480 pixel VGA monitor refreshed at 59.52 Hz, the pixel clock operates at $800 \times 525 \times 59.52 = 25$ MHz, so each pixel is 40 ns wide.

Figure e9.22(a) shows the timing of each of the scan lines. The entire scan line is 32 µs long. Figure e9.22(b) shows the vertical timing; note that the time units are now scan lines rather than pixel clocks. A new frame is drawn approximately 60 times per second. Higher resolutions use a faster pixel clock, up to 388 MHz for 2048×1536 refreshed at 85 Hz. For example, a 1024×768 display refreshed at 60 Hz can be achieved with a 65 MHz pixel clock.

Figure e9.23 shows the pinout for a female connector coming from a video source. Pixel information is conveyed with three analog voltages for red, green, and blue. Each voltage ranges from 0 to 0.7V, with more positive indicating brighter. The voltages should be 0 during the front and back porches. The video signal must be generated in real time at high speed, which is difficult on a microcontroller but easy on an FPGA. A simple black-and-white display could be produced by driving all three color pins with either 0 or 0.7V using a voltage divider connected to a digital output pin. A color monitor, on the other hand, uses a *video DAC* with three separate D/A converters to independently drive the three color pins.

Figure e9.24 shows an FPGA driving a VGA monitor through an ADV7125 triple 8-bit video DAC. The DAC receives 8 bits of R, G, and B from the FPGA. It also receives a SYNC_b signal that is driven active low whenever HSYNC or VSYNC are asserted. The video DAC

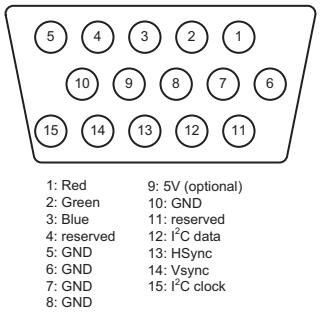


Figure e9.23 VGA connector pinout

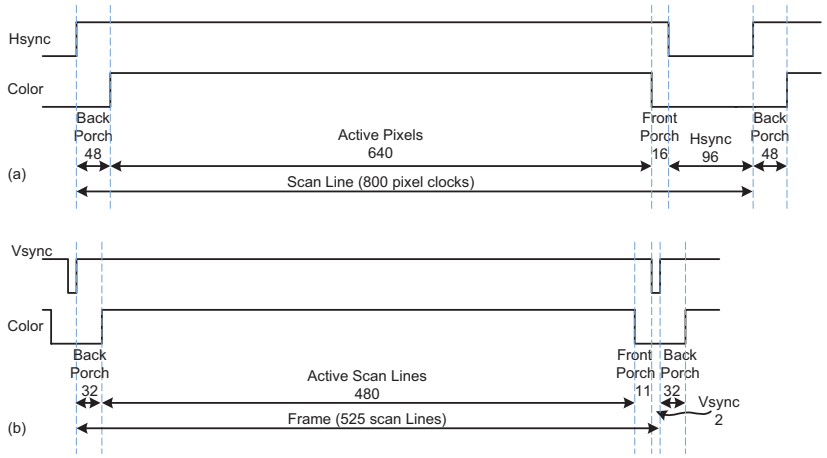


Figure e9.22 VGA timing: (a) horizontal, (b) vertical

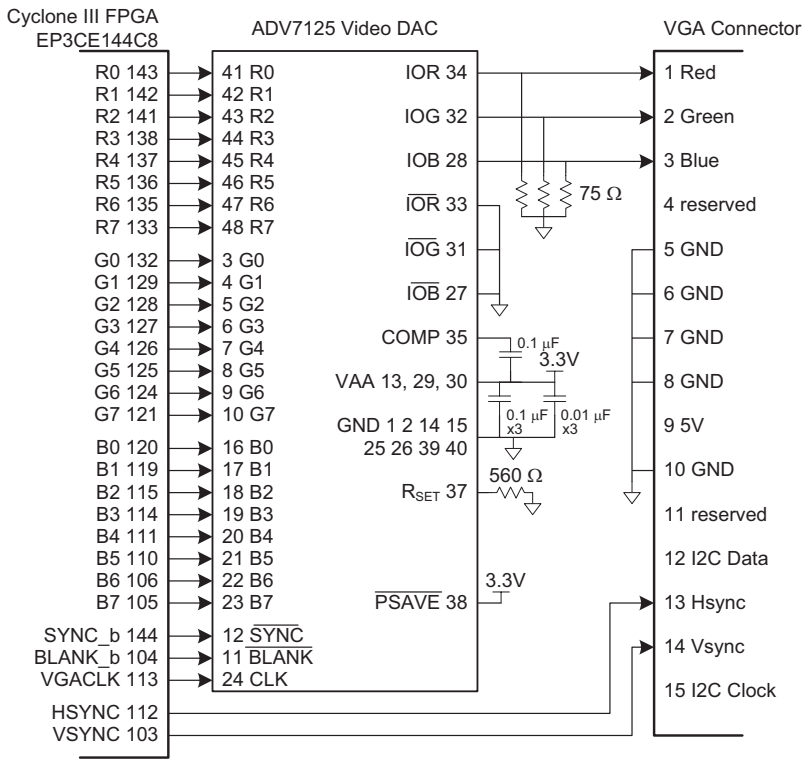


Figure e9.24 FPGA driving VGA cable through video DAC

produces three output currents to drive the red, green, and blue analog lines, which are normally 75 Ω transmission lines parallel terminated at both the video DAC and the monitor. The R_{SET} resistor sets the scale of the output current to achieve the full range of color. The clock

rate depends on the resolution and refresh rate; it may be as high as 330 MHz with a fast-grade ADV7125JSTZ330 model DAC.

Example e9.11 VGA MONITOR DISPLAY

Write HDL code to display text and a green box on a VGA monitor using the circuitry from Figure e9.24.

Solution The code assumes a system clock frequency of 50 MHz and uses a clock divider to generate the 25 MHz VGA clock. You could also use a PLL to generate the clock. PLL configuration varies among FPGAs; for the Cyclone III, the frequencies are specified with Altera's megafunction wizard. Alternatively, the VGA clock could be provided directly from a signal generator.

The VGA controller counts through the columns and rows of the screen, generating the `hsync` and `vsync` signals at the appropriate times. It also produces a `blank_b` signal that is asserted low to draw black when the coordinates are outside the 640 × 480 active region.

The video generator produces red, green, and blue color values based on the current (x, y) pixel location. (0, 0) represents the upper left corner. The generator draws a set of characters on the screen, along with a green rectangle. The character generator draws an 8 × 8-pixel character, giving a screen size of 80 × 60 characters. It looks up the character from a ROM, where it is encoded in binary as 6 columns by 8 rows. The other two columns are blank. The bit order is reversed by the SystemVerilog code because the leftmost column in the ROM file is the most significant bit, while it should be drawn in the least significant x-position.

Figure e9.25c shows a photograph of the VGA monitor while running this program. The rows of letters alternate red and blue. A green box overlays part of the image.

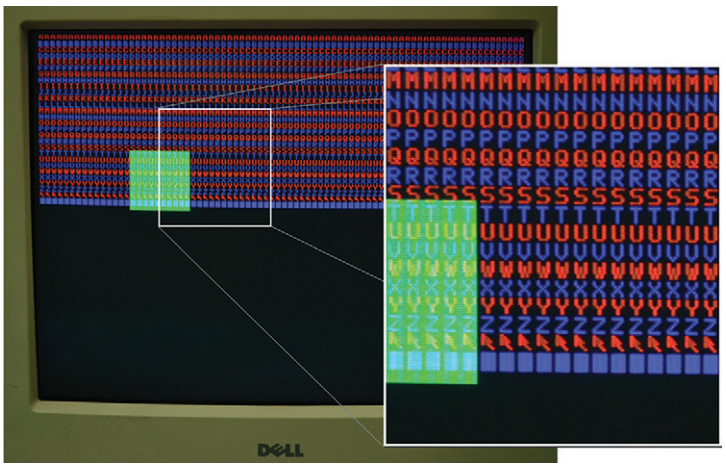


Figure e9.25 VGA output

HDL Example e9.3 vga.sv

```
module vga(input  logic clk, reset,
           output logic vgaclk,           // 25 MHz VGA clock
           output logic hsync, vsync,
           output logic sync_b, blank_b,  // to monitor & DAC
           output logic [7:0] r, g, b);   // to video DAC

  logic [9:0] x, y;

  // divide 50 MHz input clock by 2 to get 25 MHz clock
  always_ff @(posedge clk, posedge reset)
    if (reset) vgaclk = 1'b0;
    else       vgaclk = ~vgaclk;

  // generate monitor timing signals
  vgaController vgaCont(vgaclk, reset, hsync, vsync, sync_b, blank_b, x, y);

  // user-defined module to determine pixel color
  videoGen videoGen(x, y, r, g, b);

endmodule

module vgaController #(parameter HBP    = 10'd48,  // horizontal back porch
                                 HACTIVE = 10'd640, // number of pixels per line
                                 HFP    = 10'd16,  // horizontal front porch
                                 HSYN   = 10'd96,  // horizontal sync pulse = 60 to move
                                                   // electron gun back to left
                                 // number of horizontal pixels (i.e., clock cycles)
                                 HMAX   = HBP + HACTIVE + HFP + HSYN, //48+640+16+96=800:
                                 VBP    = 10'd32,  // vertical back porch
                                 VACTIVE = 10'd480, // number of lines
                                 VFP    = 10'd11,  // vertical front porch
                                 VSYN   = 10'd2,   // vertical sync pulse = 2 to move
                                                   // electron gun back to top
                                 // number of vertical pixels (i.e., clock cycles)
                                 VMAX   = VBP + VACTIVE + VFP  + VSYN) //32+480+11+2=525:

    (input  logic vgaclk, reset,
     output logic hsync, vsync, sync_b, blank_b,
     output logic [9:0] hcnt, vcnt);

     // counters for horizontal and vertical positions
     always @(posedge vgaclk, posedge reset) begin
       if (reset) begin
         hcnt <= 0;
         vcnt <= 0;
       end
       else  begin
         hcnt++;
          if (hcnt == HMAX) begin
           hcnt <= 0;
               vcnt++;
               if (vcnt == VMAX)
                 vcnt <= 0;
          end
       end
     end


     // compute sync signals (active low)
     assign hsync  = ~( (hcnt >= (HACTIVE + HFP)) & (hcnt < (HACTIVE + HFP + HSYN)) );
     assign vsync  = ~( (vcnt >= (VACTIVE + VFP)) & (vcnt < (VACTIVE + VFP + VSYN)) );
     assign sync_b = 1'b0;   // this should be 0 for newer monitors
                             // for older monitors, use: assign sync_b = hsync & vsync;
     // force outputs to black when not writing pixels
     assign blank_b = (hcnt < HACTIVE) & (vcnt < VACTIVE);
endmodule
```

```
module videoGen(input logic [9:0] x, y, output logic [7:0] r, g, b);
  logic pixel, inrect;

  // given y position, choose a character to display
  // then look up the pixel value from the character ROM
  // and display it in red or blue. Also draw a green rectangle.
  chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
  rectgen rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00, {8{pixel}}};
  assign g      = inrect   ? 8'hFF : 8'h00;
endmodule

module chargenrom(input  logic [7:0] ch,
                  input  logic [2:0] xoff, yoff,
                  output logic       pixel);

  logic [5:0] charrom[2047:0]; // character generator ROM
  logic [7:0] line;            // a line read from the ROM

  // initialize ROM with characters from text file
  initial $readmemb("charrom.txt", charrom);

  // index into ROM to find line of character
  assign line = charrom[yoff+{ch-65, 3'b000}]; // subtract 65 because A
                                               // is entry 0
  // reverse order of bits
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input  logic [9:0] x, y, left, top, right, bot,
               output logic inrect);

  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

HDL Example e9.4 charrom.txt: CONTENTS OF THE CHARACTER ROM

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
//C ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

King Bluetooth
(*So... this is actually Olof Kindgren, but we imagine King Bluetooth looked similar. Photo reprinted with permission.*)
The Bluetooth standard is named for King Harald Bluetooth of Denmark, a 10th century monarch who unified the warring Danish tribes. This wireless standard is only partially successful at unifying a host of competing wireless protocols!

Table e9.13 Bluetooth classes

Class	Transmitter Power (mW)	Range (m)
1	100	100
2	2.5	10
3	1	5

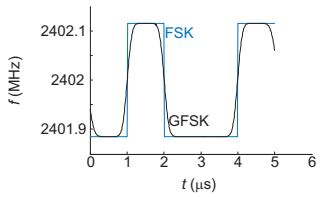


Figure e9.26 FSK and GFSK waveforms

9.4.3 Bluetooth Wireless Communication

Many standards are now available for wireless communication, including Wi-Fi, ZigBee, and Bluetooth. The standards are elaborate and require sophisticated integrated circuits, but a growing assortment of modules abstract away the complexity and give the user a simple interface for wireless communication. One of these modules is the BlueSMiRF, which is an easy-to-use Bluetooth wireless interface that can be used instead of a serial cable.

Bluetooth is a wireless standard initially developed by Ericsson in 1994 for low-power, moderate-speed communication over distances of 5 to 100 meters, depending on the transmitter power level. It is commonly used to connect an earpiece to a cellphone or a keyboard to a computer. Unlike infrared communication links, it does not require a direct line of sight between devices.

Bluetooth operates in the 2.4 GHz unlicensed industrial-scientific-medical (ISM) band. It defines 79 radio channels spaced at 1 MHz intervals starting at 2402 MHz. It hops between these channels in a pseudo-random pattern to avoid consistent interference with other devices, such as wireless routers operating in the same band. As given in Table e9.13, Bluetooth transmitters are classified at one of three power levels, which dictate the range and power consumption. In the basic rate mode, it operates at 1 Mbit/sec using Gaussian frequency shift keying (FSK). In ordinary FSK, each bit is conveyed by transmitting a frequency of $f_c \pm f_d$, where f_c is the center frequency of the channel and f_d is an offset of at least 115 kHz. The abrupt transition in frequencies between bits consumes extra bandwidth. In Gaussian FSK, the change in frequency is smoothed to make better use of the spectrum. Figure e9.26 shows the frequencies being transmitted for a sequence of 0's and 1's on a 2402 MHz channel using FSK and GFSK.

A BlueSMiRF Silver module, shown in Figure e9.27(a), contains a Class 2 Bluetooth radio, modem, and interface circuitry on a small card with a serial interface. It communicates with another Bluetooth device, such as a laptop with built-in Bluetooth, or a Bluetooth USB dongle connected to a PC. Thus, it can provide a wireless serial link between a RED-V and a PC similar to the link from Figure e9.13 but without the cable. The wireless link is compatible with the same software as is the wired link.

Figure e9.28 shows a schematic for such a link. The TX pin of the BlueSMiRF connects to the RX pin of the RED-V and vice versa. The RTS and CTS pins are connected so that the BlueSMiRF shakes its own hand.

The BlueSMiRF defaults to 115.2 kbaud with 8 data bits, 1 stop bit, and no parity or flow control. It operates at 3.3 V digital logic

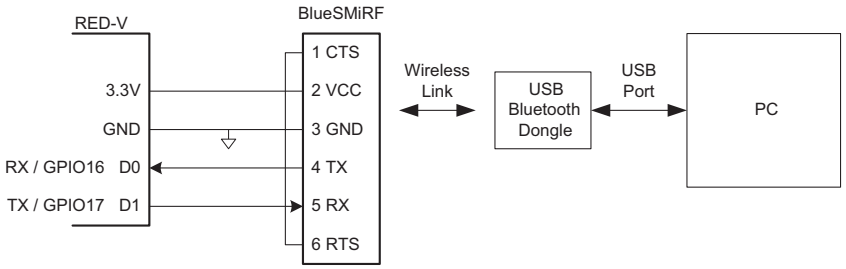


Figure e9.28 BlueSMiRF RED-V to PC link

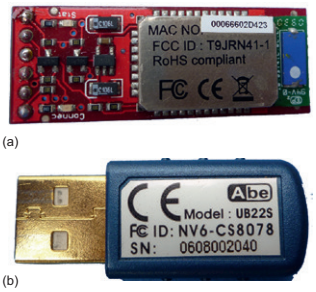


Figure e9.27 (a) BlueSMiRF module and (b) USB dongle

levels, so no RS-232 transceiver is necessary to connect with another 3.3 V device.

To use the interface, plug a USB Bluetooth dongle into a PC. Power up the RED-V and BlueSMiRF. The red STAT light will flash on the BlueSMiRF, indicating that it is waiting to make a connection. Open the Bluetooth icon in the PC system tray and use the Add Bluetooth Device Wizard to pair the dongle with the BlueSMiRF. The default passkey for the BlueSMiRF is 1234. Take note of which COM port is assigned to the dongle. Then communication can proceed just as it would over a serial cable. Note that the dongle typically operates at 9600 baud and that PuTTY must be configured accordingly.

9.4.4 Motor Control

Another major application of microcontrollers is to drive actuators such as motors. This section describes three types of motors: DC motors, servo motors, and stepper motors. *DC motors* require a high drive current, typically on the order of 1 A. Thus, a microcontroller's GPIO cannot drive them directly and a powerful driver such as an *H-bridge* must be connected between the microcontroller and the motor. Motors also require a *shaft encoder* if the user wants to know the current position of the motor. *Servo motors* accept a pulse-width modulated signal to specify their position over a limited range of angles. They are very easy to interface but are not as powerful and are not suited to continuous rotation. *Stepper motors* accept a sequence of pulses, each of which rotates the motor by a fixed angle, called a *step*. They are more expensive and still need an H-bridge to drive the high current, but the position can be precisely controlled.

Motors can draw a substantial amount of current and may introduce glitches on the power supply that disturb digital logic. One way to reduce this problem is to use a different power supply or battery for the motor than for the digital logic.

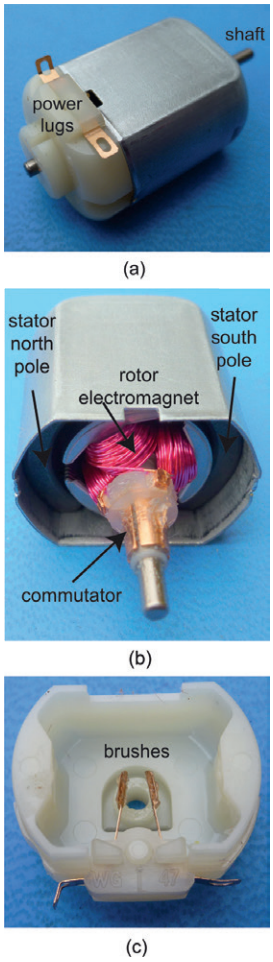


Figure e9.29 DC motor

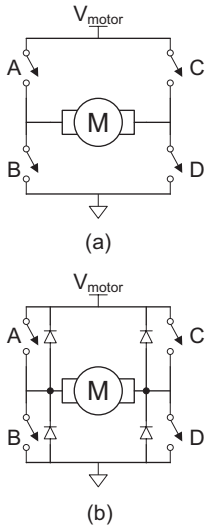


Figure e9.30 H-bridge

DC Motors

Figure e9.29 shows the structure of a brushed DC motor. The motor is a two-terminal device. It contains permanent stationary magnets called the *stator* and a rotating electromagnet called the *rotor* or *armature* connected to the shaft. The front end of the rotor connects to a split metal ring called a *commutator*. Metal brushes attached to the power lugs (input terminals) rub against the commutator, providing current to the rotor's electromagnet. This induces a magnetic field in the rotor that causes the rotor to spin to become aligned with the stator field. Once the rotor has spun part way around and approaches alignment with the stator, the brushes touch the opposite sides of the commutator, reversing the current flow and magnetic field and causing it to continue spinning indefinitely.

DC motors tend to spin at thousands of rotations per minute (RPM) at very low torque. Most systems add a gear train to reduce the speed to a more reasonable level and increase the torque. Look for a gear train designed to mate with your motor. Pittman manufactures a wide range of high-quality DC motors and accessories, while inexpensive toy motors are popular among hobbyists.

A DC motor requires substantial current and voltage to deliver significant power to a load. The current should also be reversible so the motor can spin in both directions. Most microcontrollers cannot produce enough current to drive a DC motor directly. Instead, they use an H-bridge, which conceptually contains four electrically controlled switches, as shown in Figure e9.30(a). It is called an H-bridge because the configuration of switches mimics the letter H. If switches A and D are closed, current flows from left to right through the motor and it spins in one direction. If B and C are closed, current flows from right to left through the motor and it spins in the other direction. If A and C or B and D are closed, the voltage across the motor is forced to 0, causing the motor to actively brake. If none of the switches are closed, the motor will coast to a stop. The switches in an H-bridge are *power transistors*, that is, they can carry high currents of one or more Amps. The H-bridge also contains some digital logic to conveniently control the switches. The microcontroller supplies a low-current digital input to control the H-bridge high-current output.

When the motor current changes abruptly, the inductance of the motor's electromagnet will induce a large voltage spike that could damage the power transistors. Therefore, many H-bridges also have protection diodes in parallel with the switches, as shown in Figure e9.30(b). If the inductive kick drives either terminal of the motor above V_{motor} or below ground, the diodes will turn ON and clamp the voltage at a safe level. H-bridges can dissipate large amounts of power, so a heat sink may be necessary to keep them cool.

Example e9.12 AUTONOMOUS VEHICLE

Design a system in which a RED-V board controls two drive motors for a robot car. Write a library of functions to initialize the motor driver and to make the car drive forward and back, turn left or right, and stop. Use PWM to vary the voltage output and, thus, control the speed of the motors.

Solution Figure e9.31 shows a pair of DC motors controlled by a RED-V via a Texas Instruments SN754410 dual H-bridge. The H-bridge requires a 5 V logic supply V_{CC1} and a 4.5 to 36 V motor supply V_{CC2}; it has $V_{IH} = 2\,V$ and, hence, is compatible with the 3.3 V I/O from the RED-V. It can deliver up to 1 A of current to each of two motors. V_{motor} should come from a separate battery pack; the 5 V output of the RED-V cannot supply enough current to drive most motors and the RED-V could be damaged.

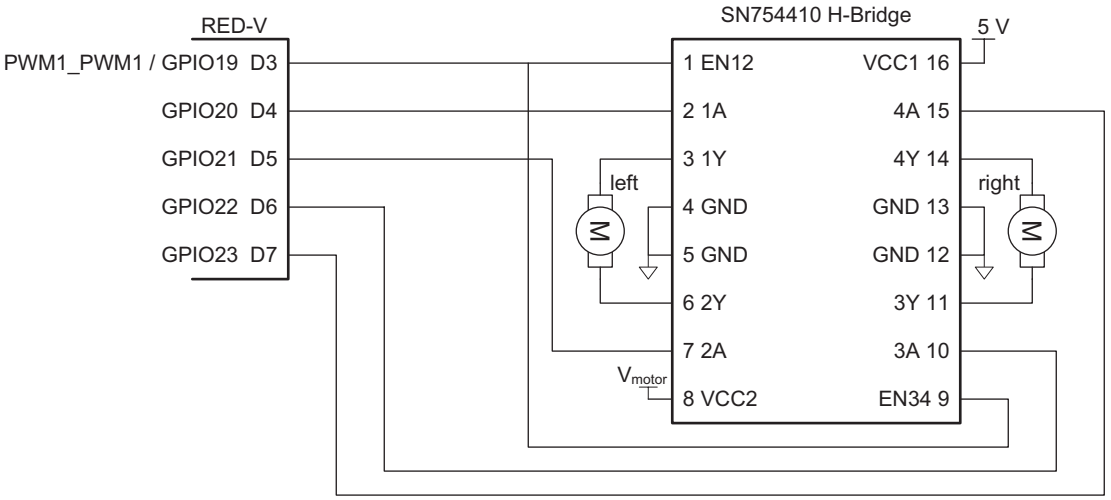


Figure e9.31 **Motor control with dual H-bridge**

Table e9.14 describes how the inputs to each H-bridge control a motor. The microcontroller drives the enable signals with a PWM signal to control the speed of the motors. It drives the four other pins to control the direction of each motor.

Table e9.14 **H-Bridge control**

EN12	1A	2A	Motor
0	X	X	Coast
1	0	0	Brake
1	0	1	Reverse
1	1	0	Forward
1	1	1	Brake

The PWM is configured to work at about 5 kHz with a duty cycle ranging from 0% to 100%. Any PWM frequency far higher than the motor's bandwidth will give the effect of smooth movement. Note that the relationship between duty cycle and motor speed is nonlinear and that below some duty cycle, the motor will not move at all.

Code Example e9.18 shows how to use PWM control with the dual H-bridge configuration shown in Figure e9.31 to drive two DC motors.

Code Example e9.18 DC MOTOR DRIVER

```
#include "EasyREDVIO.h"

// Motor Constants
#define EN D3
#define MOTOR_1A D4
#define MOTOR_2A D5
#define MOTOR_3A D6
#define MOTOR_4A D7

void setMotorLeft(int dir) {  // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_1A, dir);
    digitalWrite(MOTOR_2A, !dir);
}

void setMotorRight(int dir){  // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_3A, dir);
    digitalWrite(MOTOR_4A, !dir);
}

void forward(void) {
    setMotorLeft(1); setMotorRight(1); // both motors drive forward
}

void backward(void) {
    setMotorLeft(0); setMotorRight(0); // both motors drive backward
}

void left(void) {
    setMotorLeft(0); setMotorRight(1); // left back, right forward
}

void right(void) {
    setMotorLeft(1); setMotorRight(0); // right back, left forward
}

void halt(void) {  // turn both motors off
    digitalWrite(MOTOR_1A, 0);
    digitalWrite(MOTOR_2A, 0);
    digitalWrite(MOTOR_3A, 0);
    digitalWrite(MOTOR_4A, 0);
}

void initMotors(void) {
    pinMode(MOTOR_1A, OUTPUT);
    pinMode(MOTOR_2A, OUTPUT);
    pinMode(MOTOR_3A, OUTPUT);
    pinMode(MOTOR_4A, OUTPUT);
    halt();                     // ensure motors are not spinning
    pwmInit(EN, 1, 255);        // turn on PWM
    analogWrite(200);           // default to partial power
}

int main(void) {
    initMotors();
    while(1)
    {
        forward();
        delay(5000);
        backward();
        delay(5000);
        left();
        delay(5000);
        right();
        delay(5000);
        halt();
    }
}
```

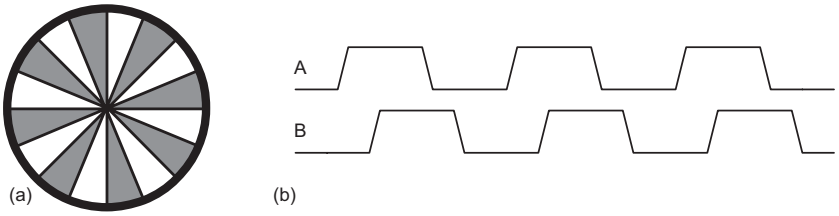


Figure e9.32 Shaft encoder (a) disk, (b) quadrature outputs

In the previous example, there is no way to measure the position of each motor. Two motors are unlikely to be exactly matched, so one is likely to turn slightly faster than the other, causing the robot to veer off course. To solve this problem, some systems add shaft encoders. Figure e9.32(a) shows a simple shaft encoder consisting of a disk with slots attached to the motor shaft. An LED is placed on one side and a light sensor is placed on the other side. The shaft encoder produces a pulse every time the gap rotates past the LED. A microcontroller can count these pulses to measure the total angle that the shaft has turned. By using two LED/sensor pairs spaced half a slot width apart, an improved shaft encoder can produce quadrature outputs shown in Figure e9.32(b) that indicate the direction the shaft is turning, as well as the angle by which it has turned. Sometimes shaft encoders add another hole to indicate when the shaft is at an index position.

Servo Motor

A servo motor is a DC motor integrated with a gear train, a shaft encoder, and some control logic so that it is easier to use. It has a limited rotation, typically 180°. Figure e9.33 shows a servo with the lid removed to reveal the gears. A servo motor has a 3-pin interface with power (typically 5 V), ground, and a control input. The control input is typically a 50 Hz pulse-width modulated signal. The servo's control logic drives the shaft to a position determined by the duty cycle of the control input. The servo's shaft encoder is typically a rotary potentiometer that produces a voltage dependent on the shaft position.

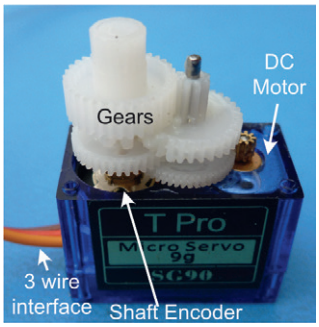


Figure e9.33 SG90 servo motor

In a typical servo motor with 180 degrees of rotation, a pulse width of 1 ms drives the shaft to 0°, 1.5 ms to 90°, and 2 ms to 180°. For example, Figure e9.34 shows a control signal with a 1.5 ms pulse width. Driving the servo outside its range may cause it to hit mechanical stops and be damaged. The servo's power comes from the power pin rather than the control pin, so the control can connect directly to a microcontroller without an H-bridge. Servo motors are commonly used in remote-control model airplanes and small robots because they are small, light, and convenient. Finding a motor with an adequate datasheet can

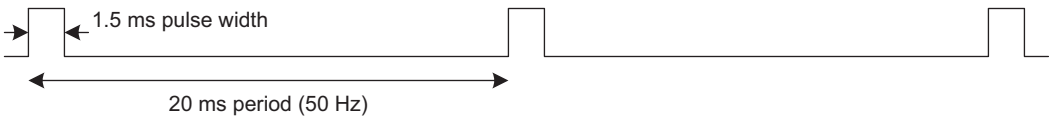


Figure e9.34 Servo control waveform

be difficult. The center pin with a red wire is normally power, and the black or brown wire is normally ground.

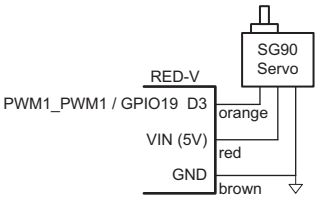


Figure e9.35 Servo motor control

Example e9.13 SERVO MOTOR

Design a system in which a RED-V drives a servo motor to a desired angle.

Solution Figure e9.35 shows a diagram of the connection to an SG90 servo motor, including the colors of the wires on the servo cable. The servo operates off of a 4.0 to 7.2 V power supply. It can draw as much as 0.5 A if it must deliver a large amount of force but may run directly off the RED-V power supply if the load is light. A single wire carries the PWM signal, which can be provided at 5 or 3.3 V logic levels. The code configures the PWM generation and computes the appropriate duty cycle for the desired angle. It cycles through positioning the servo at 0°, 90°, and 180°.

Code Example e9.19 SERVO MOTOR DRIVER

```
#include "EasyREDVIO.h"

void genPulseMicroseconds(uint16_t pulse_len_us) {
    PWM1->pwmcmp1.pwmcmp = pulse_len_us;
}

void setServo(float angle) {
    volatile uint16_t pulse_len_us = (uint16_t) (1000 + (angle / 180) * 1000);
    genPulseMicroseconds(pulse_len_us);
}

int main(void) {
    uint32_t scale = 4; // Set scale to get 16e6/2^4 = 1 MHz count speed for 1 us accuracy
    float freq = 50.0;
    volatile uint32_t pwm_period_count = (uint32_t) (1/freq * 1e6); // Period for PWM in
                                                                    // microseconds

    pwmInit(D3, scale, pwm_period_count);
    while(1) {
        setServo(0.0);
        delay(1000);
        setServo(90.0);
        delay(1000);
        setServo(180.0);
        delay(1000);
    }
}
```

It is also possible to convert an ordinary servo into a continuous rotation servo by carefully disassembling it, removing the mechanical stop, and replacing the potentiometer with a fixed voltage divider. Many websites show detailed directions for particular servos. The PWM will then control the velocity rather than position, with 1.5 ms indicating stop, 2 ms indicating full speed forward, and 1 ms indicating full speed backward. A continuous rotation servo may be more convenient and less expensive than a simple DC motor combined with an H-bridge and gear train.

Stepper Motor

A stepper motor advances in discrete steps as pulses are applied to alternate inputs. The step size is usually a few degrees, allowing precise positioning and continuous rotation. Small stepper motors generally come with two sets of coils called *phases* wired in *bipolar* or *unipolar* fashion. Bipolar motors are more powerful and less expensive for a given size but require an H-bridge driver, while unipolar motors can be driven with transistors acting as switches. This section focuses on the more efficient bipolar stepper motor.

Figure e9.36(a) shows a simplified two-phase bipolar motor with a 90-degree step size. The rotor is a permanent magnet with one north and one south pole. The stator is an electromagnet with two pairs of coils comprising the two phases. Two-phase bipolar motors thus have four terminals. Figure e9.36(b) shows a symbol for the stepper motor modeling the two coils as inductors. Practical motors add gearing to reduce the output step size and increase torque.

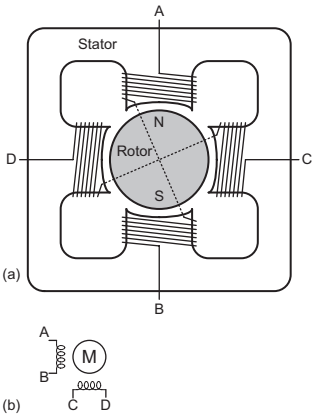


Figure e9.36 Two-phase bipolar motor: (a) simplified diagram, (b) symbol

Figure e9.37 shows three common drive sequences for a two-phase bipolar motor. Figure e9.37(a) illustrates *wave drive*, in which the coils are energized in the sequence AB–CD–BA–DC. Note that BA means that the winding AB is energized with current flowing in the opposite direction; this is the origin of the name *bipolar*. The rotor turns by 90 degrees at each step. Figure e9.37(b) illustrates *two-phase-on drive*, following the pattern (AB, CD)–(BA, CD)–(BA, DC)–(AB, DC). (AB, CD) indicates that both coils AB and CD are energized simultaneously. The rotor again turns by 90 degrees at each step, but aligns itself halfway between the two pole positions. This gives the highest torque operation because both coils are delivering power at once. Figure e9.37(c) illustrates *half-step drive*, following the pattern (AB, CD)–CD–(BA, CD)–BA–(BA, DC)–DC–(AB, DC)–AB. The rotor turns by 45 degrees at each half-step. The rate at which the pattern advances determines the speed of the motor. To reverse the motor direction, the same drive sequences are applied in the opposite order.

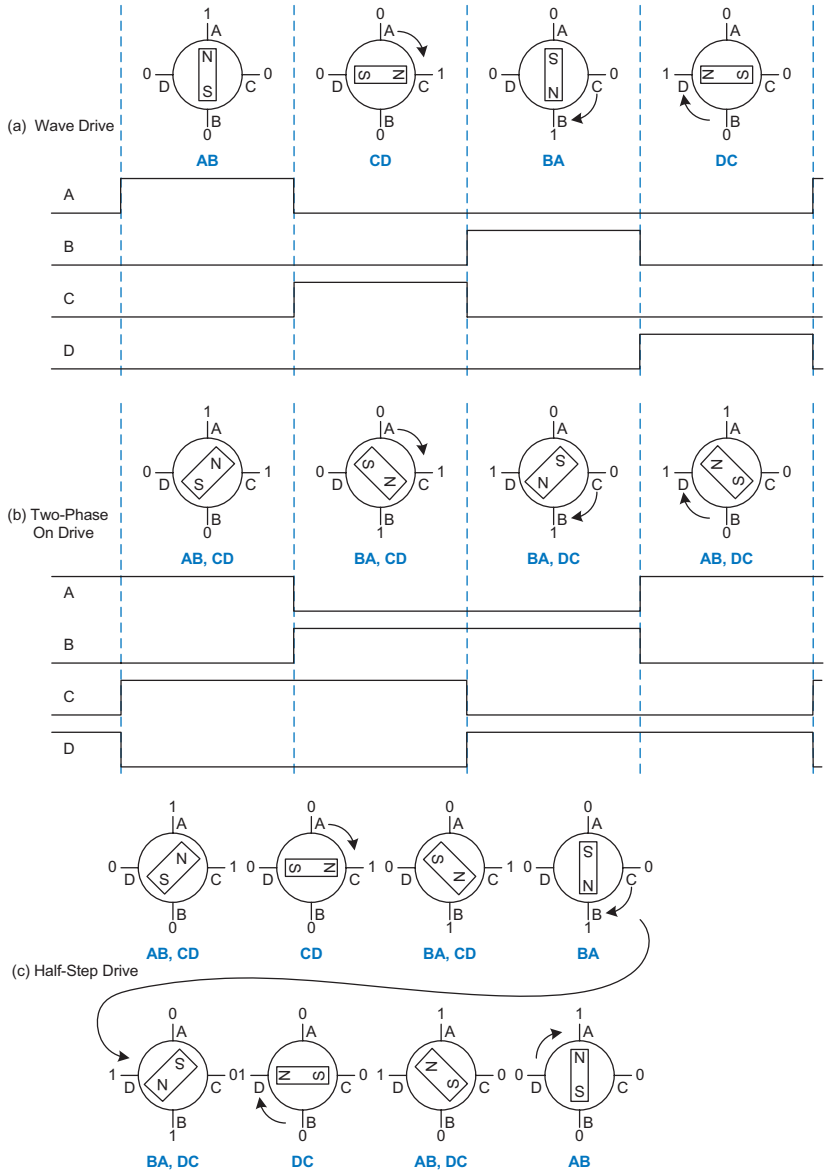


Figure e9.37 **Bipolar motor drive**

In a real motor, the rotor has many poles to make the angle between steps much smaller. For example, Figure e9.39 shows an AIRPAX LB82773-M1 bipolar stepper motor with a 7.5-degree step size. The motor operates off 5 V and draws 0.8 A through each coil.

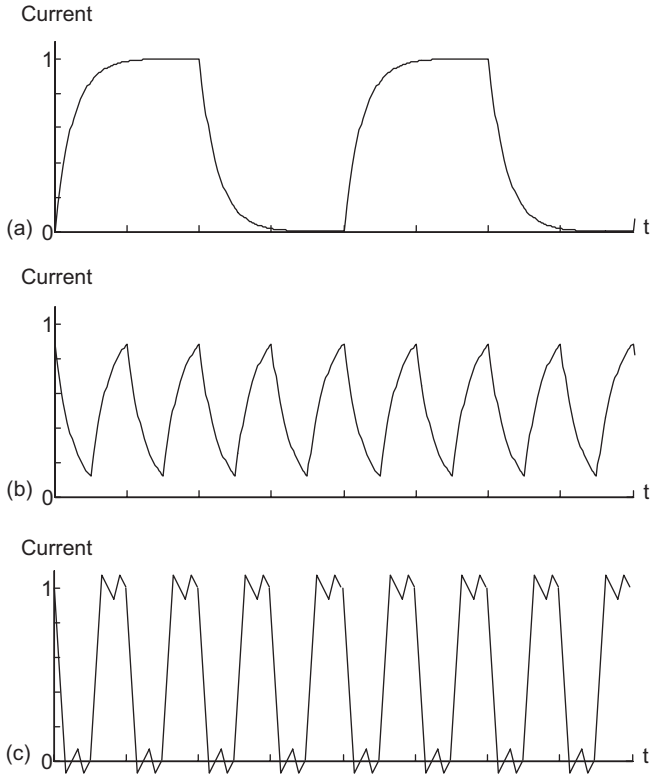


Figure e9.38 **Bipolar stepper motor direct drive current: (a) slow rotation, (b) fast rotation, (c) fast rotation with chopper drive**

The torque in the motor is proportional to the coil current. This current is determined by the voltage applied and by the inductance L and resistance R of the coil. The simplest mode of operation is called *direct voltage drive* or *L/R drive*, in which the voltage V is directly applied to the coil. The current ramps up to $I = V/R$ with a time constant set by L/R, as shown in Figure e9.38(a). This works well for slow speed operation. However, at higher speed, the current doesn't have enough time to ramp up to the full level, as shown in Figure e9.38(b), and the torque drops off.

A more efficient way to drive a stepper motor is by pulse-width modulating a higher voltage. The high voltage causes the current to ramp up to full current more rapidly; then, it is turned off (during the off portion of the PWM duty cycle) to avoid overloading the motor. The voltage is then modulated or *chopped* to maintain the current near the desired level. This is called *chopper constant current drive* and is

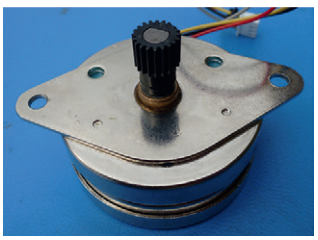


Figure e9.39 **AIRPAX LB82773-M1 bipolar stepper motor**

shown in Figure e9.38(c). The controller uses a small resistor in series with the motor to sense the current being applied by measuring the voltage drop and applies an enable signal to the H-bridge to turn off the drive when the current reaches the desired level. In principle, a microcontroller could generate the right waveforms, but it is easier to use a stepper motor controller. The L297 controller from ST Microelectronics is a convenient choice, especially when coupled with the L298 dual H-bridge with current sensing pins and a 2 A peak power capability. Unfortunately, the L298 is not available in a DIP package, so it is harder to breadboard. ST's application notes AN460 and AN470 are valuable references for stepper motor designers.

Example e9.14 BIPOLAR STEPPER MOTOR DIRECT WAVE DRIVE

Design a system to drive an AIRPAX bipolar stepper motor at a specified speed and direction using direct wave drive.

Solution Figure e9.40 shows the bipolar stepper motor driven directly by an H-bridge with the same interface as the DC motor. Note that VCC2 must supply enough voltage and current to meet the motor's demands or the motor may skip steps as the rotation rate increases.

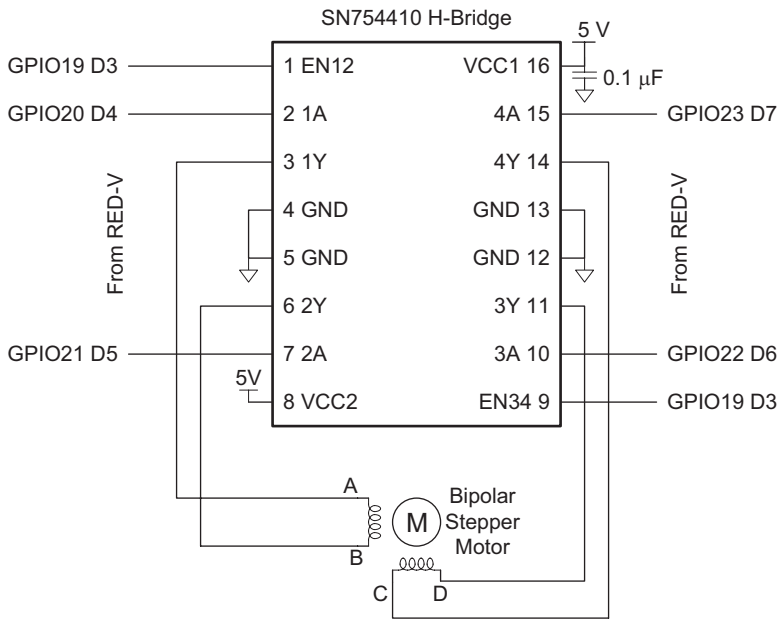


Figure e9.40 Bipolar stepper motor direct drive with H-bridge

Code Example e9.20 STEPPER MOTOR DRIVER

```
#include "EasyREDVIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MILLIS_PER_SEC 1000
#define DEG_PER_REV 360

int stepperPins[] = {19, 22, 23, 20, 21};
int curStepState; // Keep track of the current position of stepper motor

void stepperInit(void) {
  pinsMode(stepperPins, 5, OUTPUT);
  curStepState = 0;
}

void stepperSpin(int dir, int steps, float rpm) {
  int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001}; //{2A, 1A, 4A, 3A, EN}
  int step = 0;

  unsigned int millisPerStep = (SECS_PER_MIN * MILLIS_PER_SEC * STEPSIZE) /
                               (rpm * DEG_PER_REV);

  for (step = 0; step < steps; step++) {
    digitalWrites(stepperPins, 5, sequence[curStepState]);
    if (dir == 0) curStepState = (curStepState + 1) % 4;
    else  curStepState = (curStepState + 3) % 4;
    delay(millisPerStep);
  }
}
int main(void) {
  stepperInit();
  stepperSpin(1, 12000, 120); // Spin 60 revolutions at 120 rpm
}
```

9.5 SUMMARY

Most processors use memory-mapped I/O to communicate with the real world. Microcontrollers offer a range of basic peripherals including general-purpose, serial, and analog I/O and timers.

This chapter has provided many specific examples of I/O using the FE310 RISC-V microcontroller on a SparkFun RED-V RedBoard. Embedded system designers continually encounter new processors and peripherals. The general principle for incorporating simple embedded I/O is to consult the datasheet to identify the peripherals that are available and which pins and memory-mapped I/O registers are involved. Then, it is usually straightforward to write a simple device driver that initializes the peripheral's registers and transmits or receives data.

For more complex standards such as USB, writing a device driver is a highly specialized undertaking best done by an expert with detailed knowledge of the device and the USB protocol stack. Casual designers should select a processor that comes with proven device drivers and example code for the devices of interest.