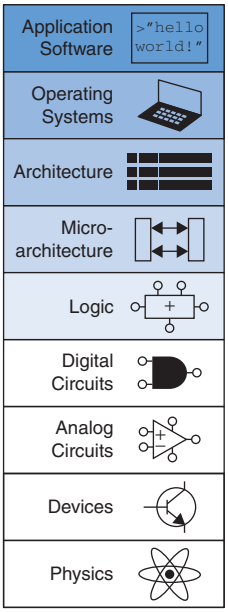
I/O Systems

e9

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, showing how to use an ARM-based Raspberry Pi single-board computer to access on-board peripherals including general-purpose, serial, and analog I/O as well as timers. Section 9.4 gives examples of interfacing with other common devices such as character LCDs, VGA monitors, Bluetooth radios, and motors. Section 9.5 describes bus interfaces and illustrates the popular AHB-Lite bus. Section 9.6 surveys the major I/O systems used in PCs.



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 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.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00015-7

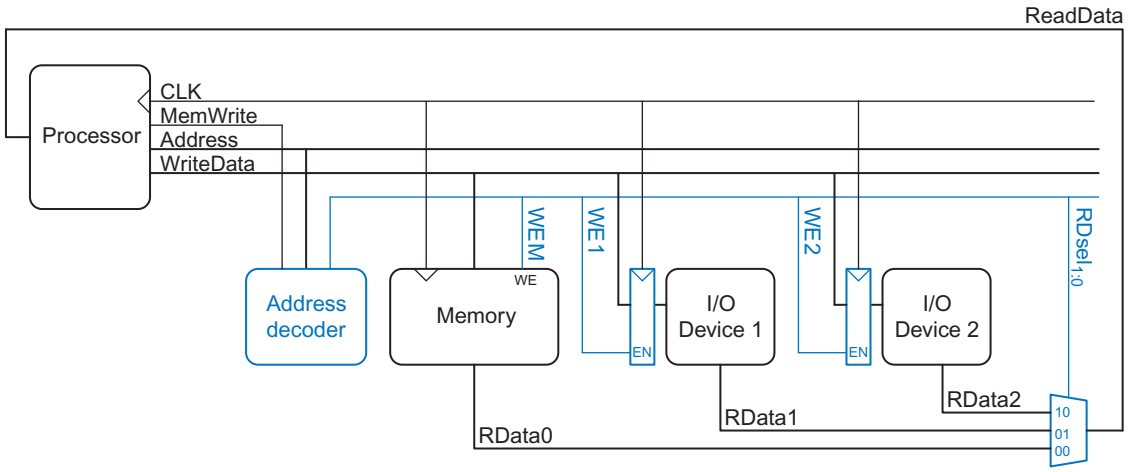


Figure e9.1 **Support hardware for memory-mapped I/O**

Example e9.1 COMMUNICATING WITH I/O DEVICES

Suppose I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the ARM 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 assembly code writes the value 7 to I/O Device 1.

```
      MOV R1, #7
      LDR R2, =ioadr
      STR R1, [R2]
ioadr DCD 0x20001000
```

The address decoder asserts *WE1* because the address is 0x20001000 and *MemWrite* is TRUE. The value on the *WriteData* bus, 7, is written into the register connected to the input pins of I/O Device 1.

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

```
  LDR R1, [R2]
```

The address decoder sets $RDsel_{1:0}$ to 01, because it detects the address 0x20001000 and *MemWrite* is FALSE. The output of I/O Device 1 passes through the multiplexer onto the *ReadData* bus and is loaded into R1 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.

Some architectures, notably x86, use specialized instructions instead of memory-mapped I/O to communicate with I/O devices. These instructions are of the following form, where `device1` and `device2` are the unique IDs of the peripheral device:

```
LDRIO R1, device1
STRIO R2, device2
```

This type of communication with I/O devices is called *programmed I/O*.

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 upon. 8-bit microcontrollers are the smallest and least expensive, while 32-bit microcontrollers provide more memory and higher performance.

Approximately $19B of microcontrollers were sold in 2014, and the market is forecast to reach $27B by 2020. The average price of a microcontroller is less than $1, and an 8-bit microcontroller can be integrated on a system-on-chip for less than a penny. Microcontrollers have become ubiquitous and nearly invisible, with an estimated 150 in each home and 50 in each automobile in 2010. The 8051 is a classic 8-bit microcontroller originally developed by Intel in 1980 and now sold by a host of manufacturers. Microchip's PIC16 and PIC18-series are 8-bit market leaders. The Atmel AVR series of microcontrollers has been popularized among hobbyists as the brain of the Arduino platform. Among 32-bit microcontrollers, Renesas leads the overall market. Freescale, Samsung, Texas Instruments, and Infineon are other major microcontroller manufacturers. ARM processors are found in nearly all smart phones and tablets today and are usually part of a system-on-chip containing the multi-core applications processor, a graphics processing unit, and extensive I/O.

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 popular and inexpensive Raspberry Pi board, which contains a Broadcom BCM2835 system-on-chip (SoC) with a 700 MHz 32-bit ARM1176JZ-F processor implementing the ARMv6 instruction set. The principles in each subsection will be followed by specific examples that run on the Pi. All of the examples have been tested on a Pi running NOOBS Raspbian Linux in 2014.

Figure e9.2 shows a photograph of a Raspberry Pi Model B+ board, which is a complete Linux computer about the size of a credit card that sells for $35. The Pi draws up to 1 A from a 5 V USB power supply. It has 512 MB of onboard RAM and an SD card socket for a memory card that contains the operating system and user files. Connectors provide video and audio output, USB ports for a mouse and keyboard, and an Ethernet (Local Area Network) port, along with 40 general-purpose I/O (GPIO) pins that are the main subject of this chapter.

While the BCM2835 SoC has many capabilities beyond those in a typical inexpensive microcontroller, the general-purpose I/O is very similar. This chapter begins by describing the BCM2835 on the Raspberry Pi and describing a device driver for memory-mapped I/O. The remainder of this chapter will illustrate how embedded systems perform general-purpose digital, analog, and serial I/O. Timers are also commonly used to generate or measure precise time intervals.

The Raspberry Pi was developed in 2011-12 by the nonprofit Raspberry Pi Foundation in the UK to promote teaching computer science. Built around the brain of an inexpensive smartphone, the computer has become wildly popular, selling more than 3 million units by 2014. The name pays homage to early home computers including Apple, Apricot, and Tangerine. Pi is derived from Python, a programming language often used in education. Documentation and purchasing information can be found at

raspberrypi.org

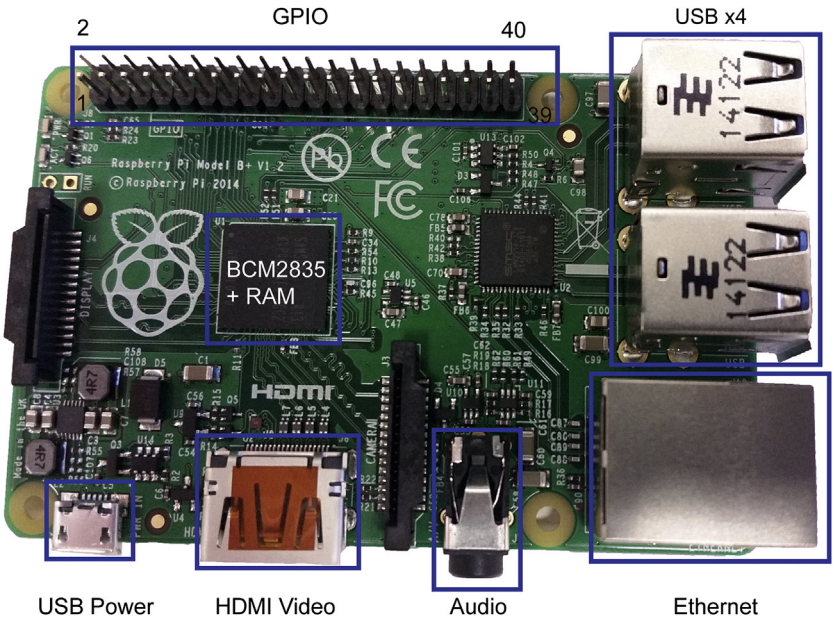


Figure e9.2 **Raspbery Pi Model B+**

Eben Upton (1978-) is the architect of the Raspberry Pi and a founder of the Raspberry Pi Foundation. He received his Bachelor's and Ph.D. from the University of Cambridge before joining Broadcom Corporation as a chip architect.

(Photograph © Eben Upton. Reproduced with permission.)

9.3.1 BCM2835 System-on-Chip

The BCM2835 SoC is a powerful yet inexpensive chip designed by Broadcom for mobile devices and other multimedia applications. The SoC includes an ARM microprocessor known as the *applications processor*, a VideoCore processor for graphics, video, and cameras, and many I/O peripherals. The BCM2835 is packaged in a plastic ball grid array with tiny solder balls underneath; it is best soldered by a robot that aligns the package to matching copper pads on a printed circuit board and applies heat. Broadcom does not publish a complete datasheet, but an abbreviated datasheet is available on the Raspberry Pi site describing how to access peripherals from the ARM processor. The datasheet describes many features and I/O registers that are omitted in this chapter for simplicity.

www.raspberrypi.org/documentation/hardware/

Figure e9.3 shows a simplified schematic of the Raspberry Pi model board. The board receives 5 V power from a USB power supply and regulators produce 3.3, 2.5, and 1.8 V levels for I/O, analog, and

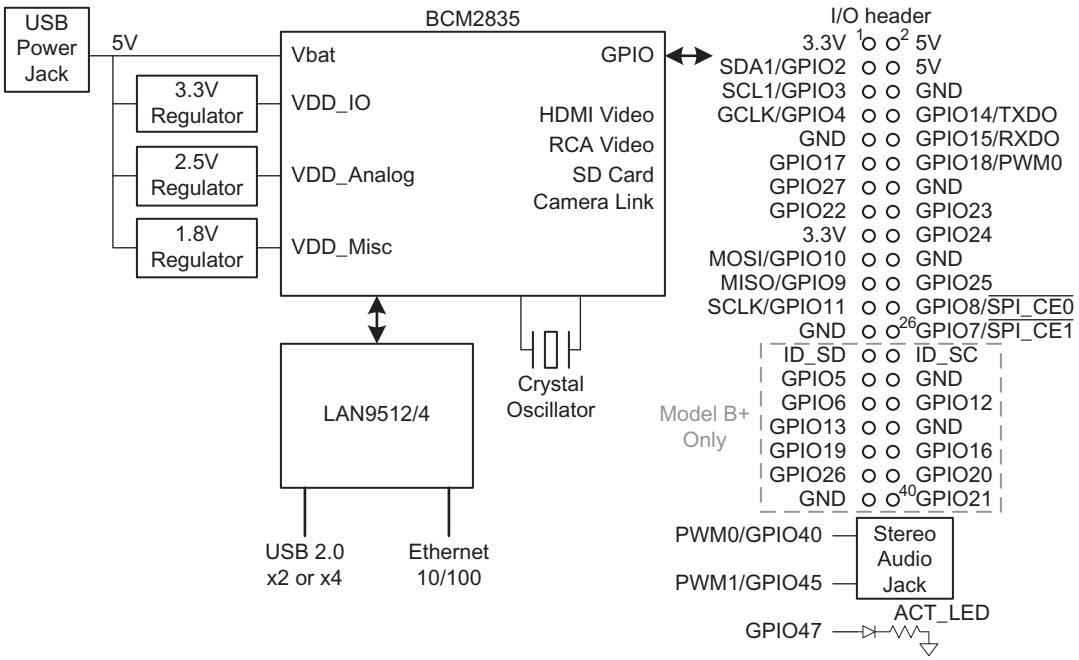


Figure e9.3 **Raspberry PI I/O schematic**

miscellaneous functions. The BCM2835 also has an internal switching regulator that produces a variable lower voltage for the power-efficient SoC. The BCM2835 connects to a USB/Ethernet controller and also directly outputs video. It also has 54 configurable I/O signals but, for space reasons, only a fraction of these are accessible to the user via header pins. The header also provides 3.3 and 5 V and ground to conveniently power small devices attached to the Pi, but the maximum total current is 50 mA from 3.3 V and ~300 mA from 5 V. The model B and B+ are similar, but B+ boosts the number of I/O header pins from 26 to 40 and the number of USB ports from 2 to 4. Various cables including the Adafruit Pi Cobbler are available to connect these header pins to a breadboard.

The Raspberry Pi uses an SD card as a Flash memory disk. The card is typically preloaded with Raspbian Linux, a small version of Linux that fits on an 8 GB SD card. You can work with the Pi either by attaching an HDMI monitor and USB mouse and keyboard to turn it into a full computer, or by connecting to it from another computer over an Ethernet cable.

The Raspberry Pi continues to advance and 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 will apply, and the principles also apply to other types of microcontrollers. You can expect to find the same types of I/O peripherals. You will need to consult the data sheet to look up the mapping between the peripheral, the pin on the chip, and the pin on the board, as well as the addresses of the memory-mapped I/O registers associated with each peripheral. You'll write configuration registers to initialize the peripheral and read and write data registers to access the peripheral.

As this book was going to press, the Raspberry Pi Foundation released the Raspberry Pi 2 Model B with a BCM2836 SoC containing a quad Cortex-A7 processor and 1 GB of RAM. The Pi 2 runs about 6 times faster than the B+ but has the same I/O as the B+ described in this chapter. The peripheral base address has moved from 0x20000000 to 0x3F000000. An updated EasyPIO supporting both models is posted to the textbook website.

Caution: the I/O connector pinout has changed between Raspberry Pi board revisions.

Caution: connecting 5 V to one of the 3.3 V I/Os will damage the I/O and possibly the entire Raspberry Pi. If you probe the I/O pins with a voltmeter, beware that you do not accidentally make contact between the 5 V pins and a nearby pin!

Pin 1 of the I/O header is labeled in Figure e9.3. When you make connections, be sure you have properly identified it and aren't rotated by 180 degrees. This is an easy mistake that could cause you to accidentally damage the Pi.

EasyPIO and the code examples in this chapter can be downloaded from the textbook website: http://booksite.elsevier.com/9780128000564. The WiringPi driver and documentation is at `wiringpi.com`.

9.3.2 Device Drivers

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.
- 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, 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 section will develop a simple device driver called EasyPIO to access BCM2835 devices so that you can understand what is happening under the hood in a device driver. Casual users are likely to prefer WiringPi, an open-source I/O library for the Pi, which has functions similar to but not exactly matching those in EasyPIO.

The memory-mapped I/O on the BCM2835 is found at physical addresses 0x20000000-0x20FFFFFF. The physical base addresses used by various peripherals are summarized in Table e9.1. Peripherals have multiple I/O registers starting at their base address. For example, reading address 0x20200034 will return the values of GPIO (general-purpose I/O) pins 31:0. The peripherals in bold will be discussed further in subsequent sections.

The Raspberry Pi typically runs a Linux operating system using virtual memory, which further complicates memory-mapped I/O. Loads and stores in a program refer to virtual addresses, not physical, so a program cannot immediately access memory-mapped I/O. Instead, it must begin by asking the operating system to map the physical addresses of interest to the program's virtual address space. The `pioInit` function from EasyPIO in Example e9.2 performs this task. The code involves some heavy duty pointer manipulation in C. The general principle is to open `/dev/mem`, which is a Linux method of accessing physical memory. Then the `mmap` function is used to set `gpio` as a pointer to physical address 0x20200000, the beginning of the GPIO registers. The pointer is declared

Table e9.1 Memory mapped I/O addresses

Physical Base Address	Peripheral
0x20003000	**System Timer**
0x2000B200	Interrupts
0x2000B400	ARM Timer
0x20200000	**GPIO**
0x20201000	**UART0**
0x20203000	PCM Audio
0x20204000	**SPI0**
0x20205000	I^2C Master #1
0x2020C000	**PWM**
0x20214000	I^2C Slave
0x20215000	miniUART1, SPI1, SPI2
0x20300000	SD Card Controller
0x20804000	I^2C Master #2
0x20805000	I^2C Master #3

`volatile`, telling the compiler that the memory-mapped I/O value might change on its own, so the program should always read the register directly instead of relying on an old value. GPLEV0 accesses the I/O register 13 words past GPIO, e.g. at 0x20200034, which contains the values of GPIO 31:0. For brevity, this example omits error checking that takes place in the actual EasyPIO library. Subsequent subsections define more registers and functions to access I/O devices.

Example e9.2 INITIALIZING MEMORY-MAPPED I/O

```
#include <sys/mman.h>
#define BCM2835_PERI_BASE  0x20000000
#define GPIO_BASE          (BCM2835_PERI_BASE + 0x200000)
volatile unsigned int      *gpio; //Pointer to base of gpio
#define GPLEV0             (*(volatile unsigned int *)(gpio + 13))
#define BLOCK_SIZE         (4*1024)
```

For security reasons, Linux only grants the *superuser* access to memory-mapped hardware. To run a program as the superuser, type `sudo` before the Linux command. The next section will give an example.

```
void pioInit(){
    int  mem_fd;
    void *reg_map;

    // /dev/mem is a psuedo-driver for accessing memory in Linux
    mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
    reg_map = mmap(
      NULL,                  // Address at which to start local mapping (null = don't-care)
      BLOCK_SIZE,            // 4KB mapped memory block
      PROT_READ|PROT_WRITE,  // Enable both reading and writing to the mapped memory
      MAP_SHARED,            // Nonexclusive access to this memory
      mem_fd,                // Map to /dev/mem
      GPIO_BASE);            // Offset to GPIO peripheral

    gpio = (volatile unsigned *)reg_map;
    close(mem_fd);
}
```

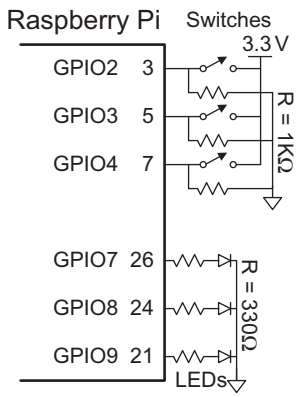


Figure e9.4 LEDs and switches connected to GPIO pins

9.3.3 General-Purpose Digital I/O

General-purpose I/O (GPIO) pins are used to read or write digital signals. For example, 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 with a 1 and to turn off when driven with a 0. The current-limiting resistors are placed in series with the LEDs to set the brightness and avoid overloading the current capability of the GPIO. The switches are wired to produce a 1 when closed and a 0 when open. The schematic indicates the pin name as well as the corresponding header pin number.

At a minimum, any GPIO pin requires registers 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 an interrupt.

Recall that the BCM2835 has 54 GPIOs. They are controlled by the GPFSEL, GPLEV, GPSET, and GPCLR registers. Figure e9.5 shows a memory map for these GPIO registers. GPFSEL5…0 determine whether each pin is a general-purpose input, output, or special-purpose I/O. Each of these function select registers uses 3 bits to specify each pin and thus

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

Table e9.2 GPFSEL register bit field to GPIO mapping

	GPFSEL0	GPFSEL1	GPFSEL2	GPFSEL3	GPFSEL4	GPFSEL5
[2:0]	GPIO0	GPIO10	GPIO20	GPIO30	GPIO40	GPIO50
[5:3]	GPIO1	GPIO11	GPIO21	GPIO31	GPIO41	GPIO51
[8:6]	GPIO2	GPIO12	GPIO22	GPIO32	GPIO42	GPIO52
[11:9]	GPIO3	GPIO13	GPIO23	GPIO33	GPIO43	GPIO53
[14:12]	GPIO4	GPIO14	GPIO24	GPIO34	GPIO44	
[17:15]	GPIO5	GPIO15	GPIO25	GPIO35	GPIO45	
[20:18]	GPIO6	GPIO16	GPIO26	GPIO36	GPIO46	
[23:21]	GPIO7	GPIO17	GPIO27	GPIO37	GPIO47	
[26:24]	GPIO8	GPIO18	GPIO28	GPIO38	GPIO48	
[29:27]	GPIO9	GPIO19	GPIO29	GPIO39	GPIO49	

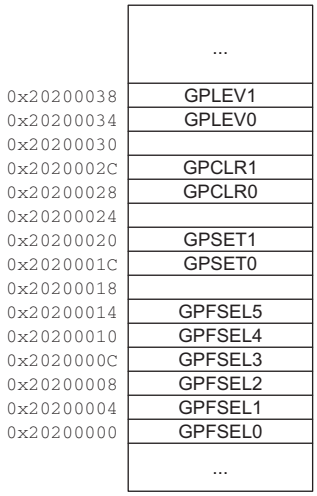


Figure e9.5 GPIO memory map

Table e9.3 GPFSEL configuration

GPFSEL	Pin Function
000	Input
001	Output
010	ALT5
011	ALT4
100	ALT0
101	ALT1
110	ALT2
111	ALT3

each 32-bit register controls 10 GPIOs as given in Table e9.2 and six GPFSEL registers are necessary to control all 54 GPIOs. For example, GPIO13 is configured by GPFSEL1[11:9]. The configurations are summarized in Table e9.3; many pins have multiple special-purpose functions that will be discussed in subsequent sections; ALT0 is most commonly used. Reading GPLEV1…0 returns the values of the pins. For example, GPIO14 is read as GPLEV0[14] and GPIO34 is read as GPLEV1[2]. The pins cannot be directly written; instead, bits are forced high or low by asserting the corresponding bit of GPSET1…0 or GPCLR1…0. For example, GPIO14 is forced to 1 by writing GPSET0[14] = 1 and forced to 0 by writing GPCLR0[14] = 1.

The BCM2835 datasheet does not specify the logic levels or output current capability of the GPIOs. However, users have determined empirically that one should not try to draw more than 16 mA from any single I/O or 50 mA total from all the I/Os. Thus, a GPIO pin is suitable for driving a small LED but not a motor. The I/Os are generally compatible with other 3.3 V chips but are not 5 V-tolerant.

The BCM2835 has unusually complex GPIO access. Some microcontrollers use a single register to configure whether each pin is input or output and another register to read and write the pins.

Example e9.3 GPIO FOR SWITCHES AND LEDS

Enhance EasyPIO with `pinMode`, `digitalRead`, and `digitalWrite` functions to configure a pin's direction and read or write it. Write a C program using these functions to read the three switches and turn on the corresponding LEDs using the hardware in Figure e9.4.

Solution: The additional EasyPIO code is given below. Because multiple registers are used to control the I/O, the functions must compute which register to access and what bit offset to use within the register. `pinMode` then clears the 0 bits and sets the 1 bits for the intended 3-bit function. `digitalWrite` handles writing either 1 or 0 by using GPSET or GPCLR. `digitalRead` pulls out the value of the desired pin and masks off the others.

```
#define GPFSEL  ((volatile unsigned int *)(gpio + 0))
#define GPSET   ((volatile unsigned int *)(gpio + 7))
#define GPCLR   ((volatile unsigned int *)(gpio + 10))
#define GPLEV   ((volatile unsigned int *)(gpio + 13))
#define INPUT   0
#define OUTPUT  1
...
void pinMode(int pin, int function) {
    int reg       =  pin/10;
    int offset    = (pin%10)*3;
    GPFSEL[reg] &= ~((0b111 & ~function) << offset);
    GPFSEL[reg] |=  ((0b111 &  function) << offset);
}

void digitalWrite(int pin, int val) {
    int reg     = pin / 32;
    int offset  = pin % 32;

    if  (val)  GPSET[reg] = 1 << offset;
    else       GPCLR[reg] = 1 << offset;
}

int digitalRead(int pin) {
    int reg     = pin / 32;
    int offset  = pin % 32;

    return (GPLEV[reg] >> offset) & 0x00000001;
}
```

The program to read switches and write LEDs is given below. It initializes GPIO access, then sets pins 2–4 as inputs for the switches and pins 7–9 as outputs for the LEDs. It then continuously reads the switches and writes their values to the corresponding LEDs.

```
#include "EasyPIO.h"

void main(void) {
    pioInit();
```

```
    // Set GPIO 4:2 as inputs
    pinMode(2, INPUT);
    pinMode(3, INPUT);
    pinMode(4, INPUT);

    // Set GPIO 9:7 as an output
    pinMode(7, OUTPUT);
    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);

    while (1) { // Read each switch and write corresponding LED
        digitalWrite(7, digitalRead(2));
        digitalWrite(8, digitalRead(3));
        digitalWrite(9, digitalRead(4));
    }
}
```

Assuming the program is in a file named dip2led.c and that EasyPIO.h is in the same directory, you can compile and run the program using the following commands on the Raspberry Pi command line . `gcc` is the C compiler. Note that `sudo` is required so that the program can access the protected I/O memory. To stop a running program, press Ctrl-C.

```
gcc dip2led.c -o dip2led
sudo ./dip2led
```

9.3.4 Serial I/O

If a microcontroller needs to send more bits than the number of free GPIO pins, it must break the message into multiple smaller transmissions. In each step, it can send either one bit or several bits. The former is called serial I/O and the latter is called parallel I/O. Serial I/O is popular 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 Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver/Transmitter (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 described in Sections 9.6.1 and 9.6.4, respectively. All five of these standards are supported on the Raspberry Pi.

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 master only needs to send data to the slave, the byte received from the slave can be ignored. If the master only needs to receive data from the slave, it must still trigger the SPI communication by sending an arbitrary byte that the slave will ignore. It can then read the data received from the slave. The SPI clock only toggles while the master is transmitting data.

9.3.4.1 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), Master Out Slave In (MOSI, also known as SDO), and Master In Slave Out (MISO, also known as SDI). SPI connects a *master* device to a *slave* device, as shown in Figure e9.6(a). The master produces the clock. It initiates communication by sending a series of clock pulses on SCK. If it wants to send data to the slave, it puts the data on MOSI, starting with the most significant bit. The slave may simultaneously respond by putting data on MISO. 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.

The BCM2835 has three SPI master ports and one slave port. This section describes SPI Master Port 0, which is readily accessible on the Raspberry Pi on GPIO pins 11:9. To use these pins for SPI rather than GPIO, their GPFSEL must be set to ALT0. The Pi must then configure the port. When the Pi writes to the SPI, the data is transmitted serially to the slave. Simultaneously, data received from the slave is collected and the Pi can read it when the transfer is complete.

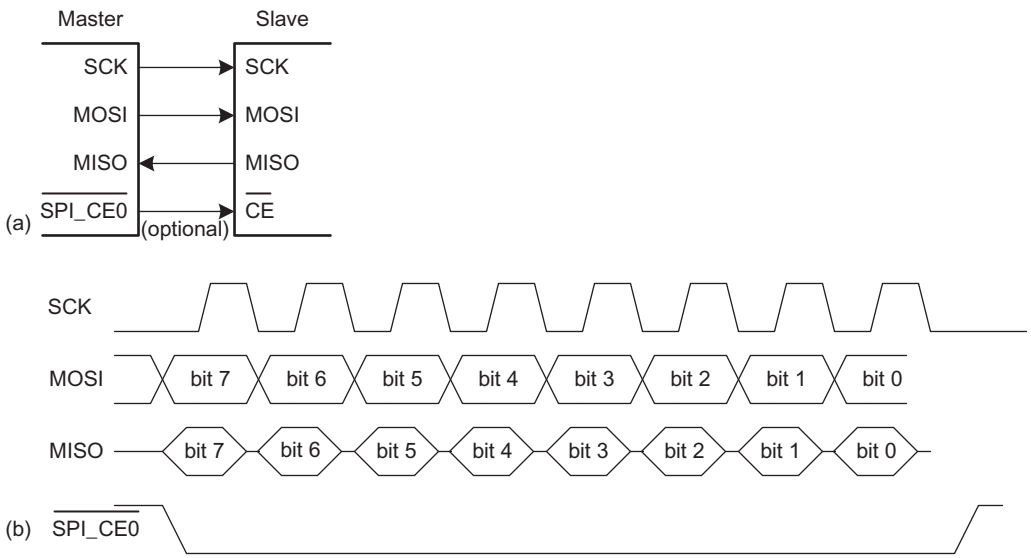


Figure e9.6 SPI connection and waveforms

Table e9.4 SPI0CS register fields

Bit	Name	Function	Meaning for 0	Meaning for 1
16	DONE	Transfer Done	Transfer in progress	Transfer complete
7	TA	Transfer Active	SPI disabled	SPI enabled
3	CPOL	Clock Polarity	Clock idles low	Clock idles high
2	CPHA	Clock Phase	First SCK transition at middle of data bit	First SCK transition at beginning of data bit

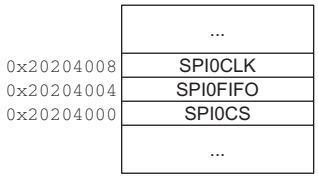


Figure e9.7 SPI Master Port 0 registers

SPI Master Port 0 is associated with three registers, given in the memory map in Figure e9.7. SPI0CS is the control register. It is used to turn the SPI on and set attributes such as the polarity of the clock. Table e9.4 lists the names and functions of some of the bits in SPI0CS that are relevant to this discussion. All have a default value of 0 on reset. Most of the functions, such as chip selects and interrupts, are not used in this section but can be found in the datasheet. SPI0FIFO is written to transmit a byte and read to get the byte received back. SPI0CLK configures the SPI clock frequency by dividing the 250 MHz peripheral clock by a power of two specified in the register. Thus, the SPI clock frequency is summarized in Table e9.5.

If the frequency is too high (>~1 MHz on a breadboard or tens of MHz on an unterminated printed circuit board), the SPI may become unreliable due to reflections, crosstalk, or other signal integrity issues.

Table e9.5 SPI0CLK frequencies

SPI0CLK	SPI Frequency (kHz)
2	125000
4	62500
8	31250
16	15625
32	7812
64	3906
128	1953
256	976
512	488
1024	244
2048	122

Example e9.4 SENDING AND RECEIVING BYTES OVER SPI

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

Solution: Figure e9.8 shows the connection between the devices using SPI Master Port 0. The pin numbers are obtained from the component datasheets (e.g., Figure e9.3). 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 is enabled, these pins cannot be used for GPIO.

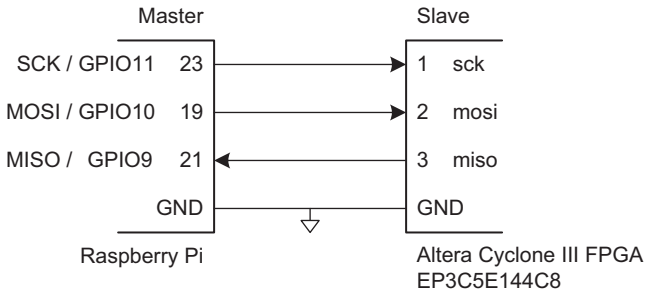


Figure e9.8 SPI connection between Pi and FPGA

The following code from EasyPIO.h is used to initialize the SPI and to send and receive a character. The code to set up the memory map and define the register addresses is similar to that for GPIO and is not reprinted here.

```
void spiInit(int freq, int settings) {
    pinMode(8, ALT0);           // CE0b
    pinMode(9, ALT0);           // MISO
    pinMode(10, ALT0);          // MOSI
    pinMode(11, ALT0);          // SCLK

    SPIOCLK = 250000000/freq; // Set SPI clock divider to desired
                                 freq
    SPIOCS = settings;
    SPIOCSbits.TA = 1;          // Turn SPI on
}

char spiSendReceive(char send){
    SPIOFIFO = send;            // Send data to slave
    while (!SPIOCSbits.DONE);   // Wait until SPI complete
    return SPIOFIFO;            // Return received data
}
```

The C code below initializes the SPI and then sends and receives a character. It sets the SPI clock to 244 kHz.

```
#include "EasyPIO.h"

void main(void) {
    char received;

    pioInit();
    spiInit(244000, 0);                 // Initialize the SPI:
                                        // 244 kHz clk, default settings
    received = spiSendReceive('A');  // Send letter A and receive byte
}
```

The HDL code for the FPGA is listed below. Figure e9.9 shows a block diagram and timing. The FPGA uses a shift register to hold the bits that have been received from the master and the bits that remain to be sent to the master.

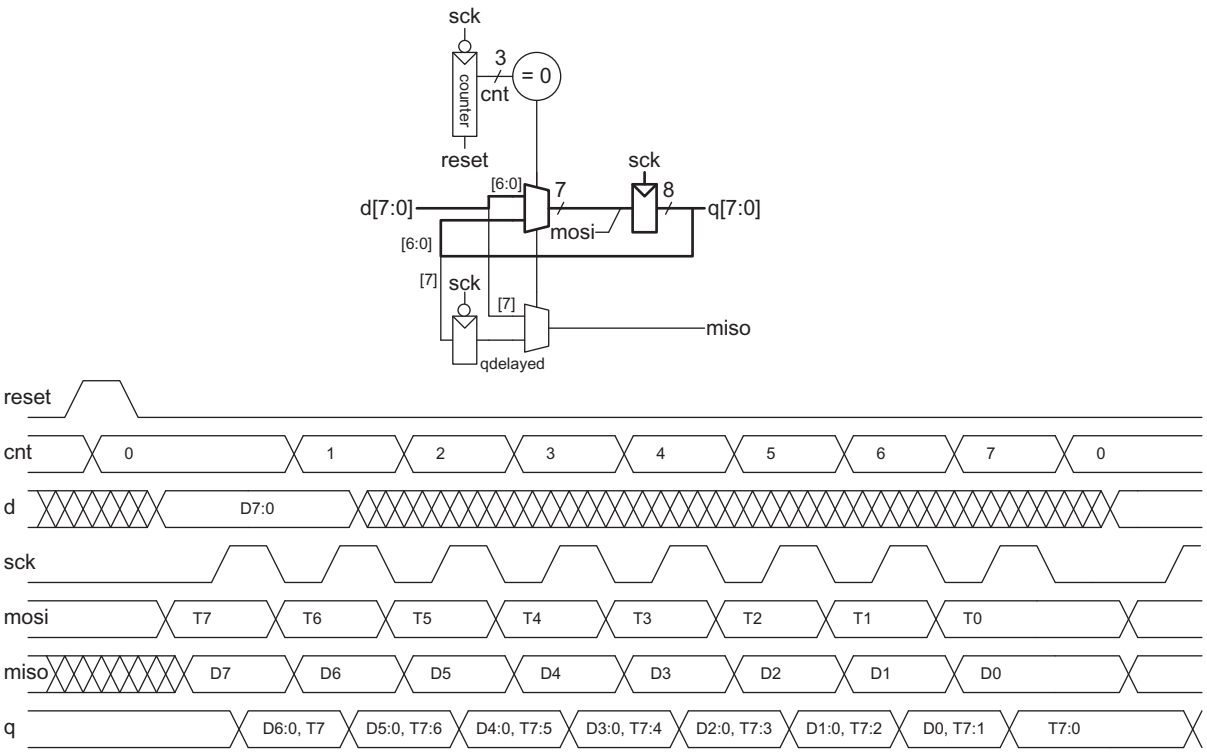


Figure e9.9 **SPI slave circuitry and timing**

On the first rising `sck` edge after reset and each 8 cycles thereafter, a new byte from `d` is loaded into the shift register. On each subsequent cycle, a bit is shifted in on `mosi` and a bit is shifted out of `miso`. `miso` is delayed until the falling edge of `sck` so that it can be sampled by the master on the next rising edge. After 8 cycles, the byte received can be found in `q`.

```
module spi_slave(input  logic       sck,     // From master
                 input  logic       mosi,    // From master
                 output logic       miso,    // To master
                 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 mosi into bottom on each step
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

  // Align miso to falling edge of sck
  // Load d at the start
  always_ff @(negedge sck)
    qdelayed = q[7];
  assign miso = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

If the slave only needs to receive data from the master, it reduces to a simple shift register given in the following HDL code.

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

SPI ports are highly configurable so that they can talk to a wide variety of serial devices. Unfortunately, this leads to the possibility of incorrectly configuring the port and garbling the data transmission. Sometimes it is necessary to change the configuration bits to communicate with a device that expects different timing. When CPOL = 1, SCK is inverted. When CPHA = 1, the clocks toggle half a cycle earlier relative to the data. These modes are shown in Figure e9.10. Be aware that different SPI products may use different names and polarities for these options; check the waveforms carefully for your device. It can also be helpful to examine

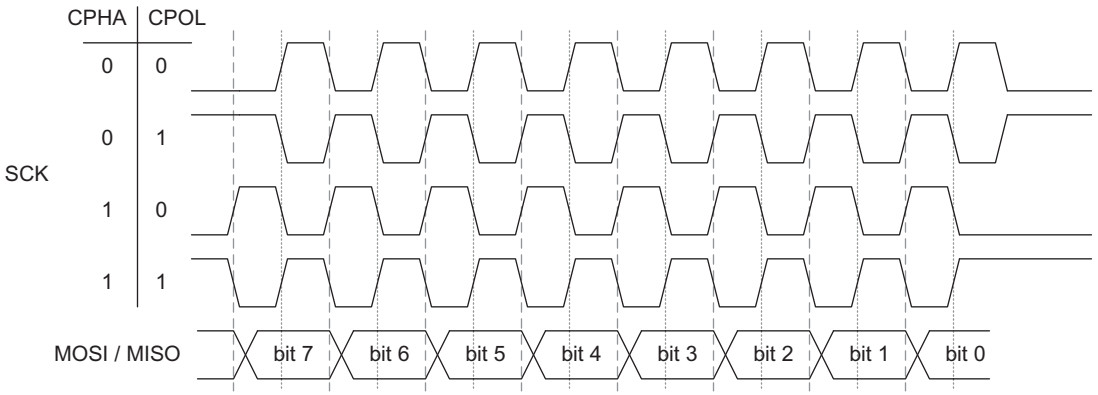


Figure e9.10 SPI clock and data timing configurations

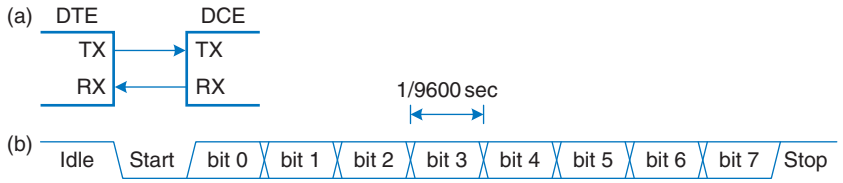


Figure e9.11 Asynchronous serial link

SCK, MOSI, and MISO on an oscilloscope if you are having communication difficulties.

9.3.4.2 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 its own clock. 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 to SPI and prone to misconfiguration issues, the standards have been around for so long that they remain important today.

Figure e9.11(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 e9.11(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). 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.

Baud rate gives the signaling rate, measured in symbols per second, whereas bit rate gives the data rate, measured in bits per second. The signaling we've discussed in this text is 2-level signaling, where each symbol represents a bit. However, multi-level signaling can send multiple bits per symbol; for example, 4-level signaling sends two bits per symbol. In that case, the bit rate is twice the baud rate. In a simple system like SPI where each symbol is a bit and each symbol represents data, 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 two-level signaling system that adds start and stop bits for each 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.

In the 1950s through 1970s, 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 e9.12 could be exploited to place free long-distance and international calls.

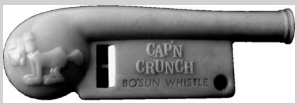


Figure e9.12 Cap'n Crunch Bosun Whistle
(Photograph by Evrim Sen, reprinted with permission.)

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.

A common choice is 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/sec,. Both systems 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 the Universal Serial Bus (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 common baud rates; 9600 is encountered where speed doesn't matter, and 115200 is the fastest standard rate, though still slow compared to 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. 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 use Data Terminal Ready (DTR), Data Carrier Detect (DCD), Data Set Ready (DSR), and Ring Indicator (RI) to indicate when equipment is connected to the line.

The original standard recommended a massive 25-pin DB-25 connector, but PCs streamlined to a male 9-pin DE-9 connector with the pinout shown in Figure e9.13(a). The cable wires normally connect straight across as shown in Figure e9.13(b). However, when directly connecting two DTEs, a *null modem* cable shown in Figure e9.13(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 guess-work 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.

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 BCM2835 has two UARTs named UART0 and UART1. Either can be configured to communicate on pins 14 and 15, but UART0 is more fully featured and is described here. To use these pins for UART0 rather than GPIO, their GPFSEL must be set to ALT0. As with SPI, the Pi 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 Figure e9.14.

To configure the UART, first set the baud rate. The UART has an internal 3 MHz clock that must be divided down to produce a clock that is 16x the desired baud rate. Hence, the appropriate divisor, BRD, is

$$\text{BRD} = 3000000/(16 \times \text{baud rate})$$

BRD is represented with a 16-bit integer portion in UART_IBRD and a 6-bit fractional portion in UART_FBRD: BRD = IBRD + FBRD/64. Table e9.6 shows these settings for popular baud rates.[1]

Table e9.6 BRD settings

Target Baud Rate	UART_IBRD	UART_FBRD	Actual Baud Rate	Error (%)
300	625	0	300	0
1200	156	16	1200	0
2400	78	8	2400	0
9600	19	34	9600	0
19200	9	49	19200	0
38400	4	56	38461	0.16
57600	3	16	57692	0.16
115200	1	40	115384	0.16

[1] The baud rates do not all evenly divide 3 MHz, so some divisors produce a frequency error. The UART, by its asynchronous nature, accommodates this error so long as it is small enough.

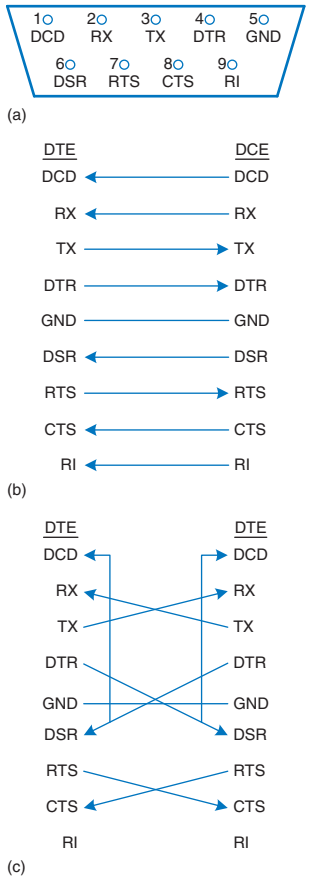


Figure e9.13 DE-9 male cable (a) pinout, (b) standard wiring, and (c) null modem wiring

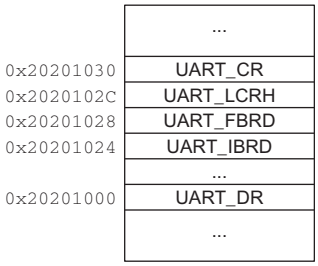


Figure e9.14 UART0 registers

Next, set the number of data, stop, and parity bits using the UART_LCRH line control register. By default, the UART has 1 stop bit and no parity, but strangely only transmits and receives 5-bit words. Hence, the WLEN field (bits 6:5) of UART_LCRH must be set to 3 to handle 8-bit words. Finally, enable the UART by turning on bit 0 (UARTEN) of the UART_CR control register.

Data is transmitted and received using the UART_DR data register and UART_FR framing register. To transmit data, wait until bit 7 (TXFE) of UART_FR is 1 to indicate that the transmitter is not busy, then write a byte to UART_DR. To receive data, wait until bit 4 (RXFE) of UART_FR is 0 to indicate that the receiver has data, then read the byte from UART_DR.

Example e9.5 SERIAL COMMUNICATION WITH A PC

Develop a circuit and a C program for a Raspberry Pi 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[2] to read and write over the serial port. The program should ask the user to type a string. It should then tell the user what she typed.

Solution: Figure e9.15(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 e9.16 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 converts the voltages from the bipolar RS-232 levels to the Pi's 3.3 V level. The Pi and PC are both Data Terminal Equipment, so the TX and RX pins must be cross-connected in the circuit. The RTS/CTS handshaking from the Pi 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 e9.15(b) shows an easier approach with an Adafruit 954 USB to TTL serial cable. The cable is directly compatible with 3.3 V levels and has female header pins that plug directly into the Raspberry Pi male headers.

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 Windows, this can be found

---

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

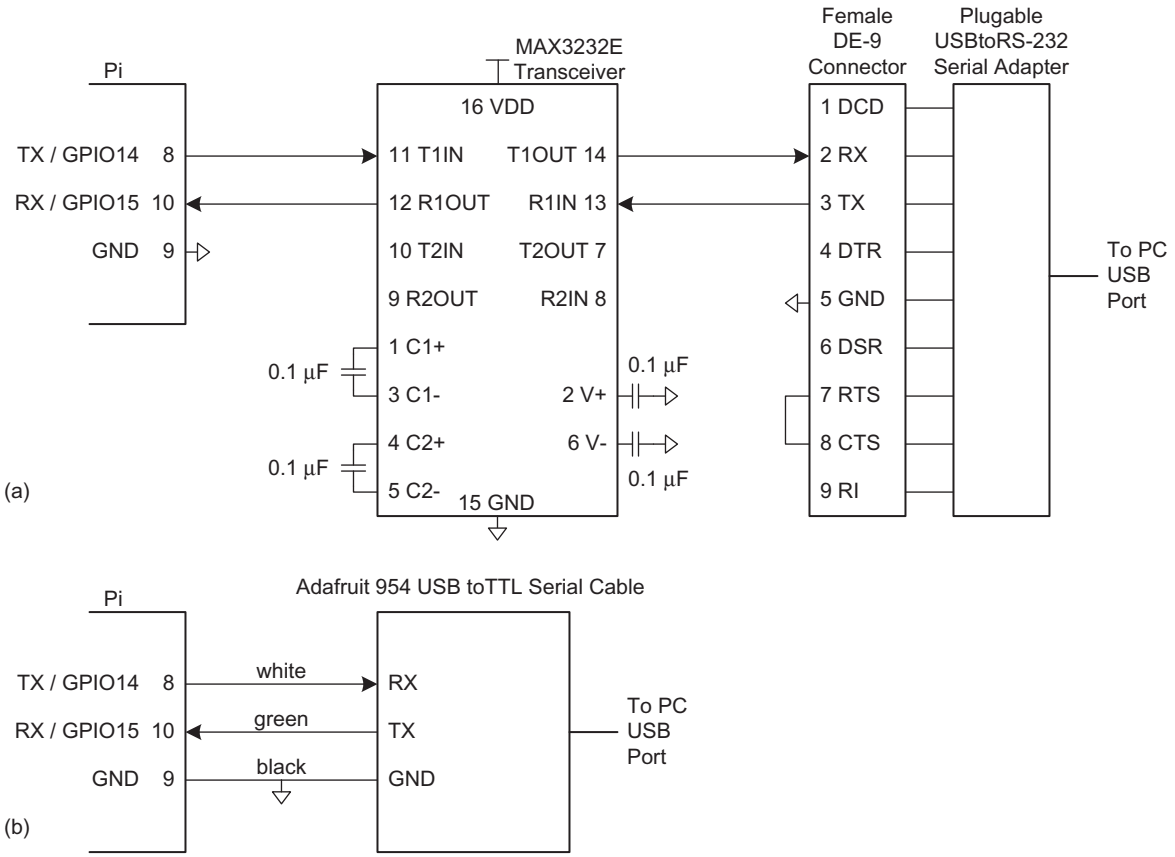


Figure e9.15 **Raspberry Pi to PC serial link (a) Plugable cable, (b) Adafruit cable**

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 EasyPIO.h is listed below. 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.[3] The `uartInit` function configures the UART as described above. Similarly, `getCharSerial` and `putCharSerial`

Note that the operating system also prints a log-in prompt to the serial port. You may see some interesting interactions between the OS and your program when both use the port.

Figure e9.16 **Plugable USB to RS-232 DB9 serial adapter**
(© 2012 Plugable Technologies; reprinted with permission.)

wait until the UART is ready and then read or write a byte from the data register.

```
void uartInit(int baud) {
    uint fb = 12000000/baud;        // 3 MHz UART clock

    pinMode(14, ALT0);              // TX
    pinMode(15, ALT0);              // RX
    UART_IBRD = fb >> 6;            // 6 Fract, 16 Int bits of BRD
    UART_FBRD = fb & 63;
    UART_LCRHbits.WLEN = 3;         // 8 Data, 1 Stop, no Parity, no FIFO, no Flow
    UART_CRbits.UARTEN = 1;         // Enable uart
}

char getCharSerial(void) {
    while (UART_FRbits.RXFE);       // Wait until data is available
    return UART_DRbits.DATA;        // Return char from serial port
}

void putCharSerial(char c) {
   while (!UART_FRbits.TXFE);       // Wait until ready to transmit
   UART_DRbits.DATA = c;            // Send char to serial port
}
```

The `main` function demonstrates printing to the console and reading from the console using the `putStrSerial` and `getStrSerial` functions.

```
#include "EasyPIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
    int i = 0;
    do {                                                    // Read an entire string until
        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];
```

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

```
    pioInit();
    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.

9.3.5 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 BCM2835 has a system timer with a 64-bit free-running counter that increments every microsecond (i.e. at 1 MHz) and four 32-bit timer compare channels. Figure e9.17 shows the memory map for the system timer. SYS_TIMER_CLO and CHI contain the lower and upper 32 bits of the 64-bit counter value. SYS_TIMER_C0…C3 are 32-bit compare channels. When any of the compare channels match SYS_TIMER_CLO, the corresponding match bit (M0-M3) in the bottom four bits of SYS_TIMER_CS is set. A match bit is cleared by writing a 1 to that bit of SYS_TIMER_CS. This may seem counterintuitive, but it prevents inadvertently clearing other match bits. One can measure a particular number of microseconds by adding that time to CLO and putting it in C1, clearing SYS_TIMER_CS.M1, then waiting until SYS_TIMER_CS.M1 is set.

Unfortunately, Linux is a multitasking operating system that may switch between processes without warning. If your program is waiting for a timer match and then another process begins executing, your program may not resume until long after the match occurs and

The graphics processing unit and operating system may use channels 0, 2, and 3, so user code should check SYSTEM_TIMER_C1.

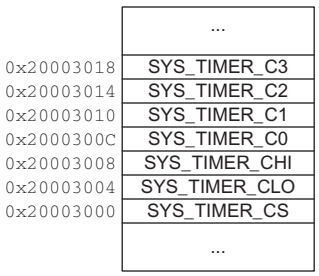


Figure e9.17 System timer registers

you may measure the wrong amount of time. To avoid this, your program can turn off interrupts during critical timing loops so that Linux will not switch processes. Be sure to turn the interrupts back on when you are done. EasyPIO defines `noInterrupts` and `interrupts` functions to disable and enable interrupts, respectively. While interrupts are disabled, the Pi will not switch between processes and cannot even respond to the user pressing Ctrl-C to kill a program. If your program hangs, you'll need to turn off power and reboot your Pi to recover.

Example e9.6 BLINKING LED

Write a program that blinks the status LED on the Raspberry Pi 5 times per second for 4 seconds.

Solution: The `delayMicros` function in EasyPIO creates a delay of a specified number of microseconds using the timer compare channel 1.

```
void delayMicros(int micros) {
    SYS_TIMER_C1 = SYS_TIMER_CLO + micros;  // Set the compare register
    SYS_TIMER_CSbits.M1 = 1;                // Reset match flag to 0
    while (SYS_TIMER_CSbits.M1 == 0);       // Wait until match flag is set
}

void delayMillis(int millis) {
    delayMicros(millis*1000);               // 1000 µs per ms
}
```

GPIO47 drives the activity LED on the Pi B+. The program sets this pin to be an output and disables interrupts. It then turns the LED OFF and ON through a series of digital writes with a 200 ms repetition rate (5 Hz). The program finally reenables interrupts.

```
#include "EasyPIO.h"

void main(void) {
    int i;

    pioInit();

    pinMode(47, OUTPUT);     // Status led as output
    noInterrupts();          // Disable interrupts

    for (i=0; i<20; i++) {
        delayMillis(150);
        digitalWrite(47, 0); // Turn led off
        delayMillis(50);
        digitalWrite(47, 1); // Turn led on
    }
    interrupts();            // Re-enable interrupts
}
```

9.3.6 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 e9.18 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 0–5 V with a sampling rate of $f_s = 44$ kHz and an accuracy of ±3 least significant bits (lsbs). Sampling rates are also listed in samples per second (sps), where 1 sps = 1 Hz. The relationship between the analog input voltage $V_{in}(t)$ and the digital sample $X[n = t / f_s]$ is

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

For example, an input voltage of 2.5 V (half of full scale) would correspond to an output of $100000000000_2 = 800_{16}$, 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

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

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 may also be used. However, microcontrollers often produce analog outputs using a technique called pulse-width modulation (PWM).

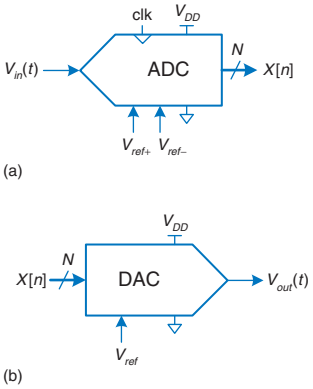


Figure e9.18 ADC and DAC symbols

9.3.6.1 D/A Conversion

The BCM2835 has a specialized DAC for composite video output, but no general-purpose converter. This section describes D/A conversion using external DACs and illustrates interfacing the Raspberry Pi to other chips over parallel and serial ports. The next section achieves the same result using pulse-width modulation.

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 Analog Devices AD558 8-bit parallel DAC and the Linear Technology LTC1257 12-bit serial DAC. Both produce voltage outputs, run off a single 5–15 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 AD558 produces an output on a scale of 0-2.56 V, consumes 75 mW, comes in a 16-pin package, and has a 1 µs settling time permitting an output rate of 1 Msamples/sec. The datasheet is at analog.com. The LTC1257 produces an output on a scale of 0–2.048 V, consumes less than 2 mW, comes in an 8-pin package, and has a 6 µs settling time. Its SPI operates at a maximum of 1.4 MHz. The datasheet is at linear.com.

Example e9.7 ANALOG OUTPUT WITH EXTERNAL DACS

Sketch a circuit and write the software for a simple signal generator producing sine and triangle waves using a Raspberry Pi, an AD558, and an LTC1257.

Solution: The circuit is shown in Figure e9.19. The AD558 connects to the Pi via GPIO14, 15, 17, 18, 22, 23, 24, and 25. It connects *Vout Sense* and *Vout Select* to *Vout* to set the 2.56 V full-scale output range. The LTC1257 connects

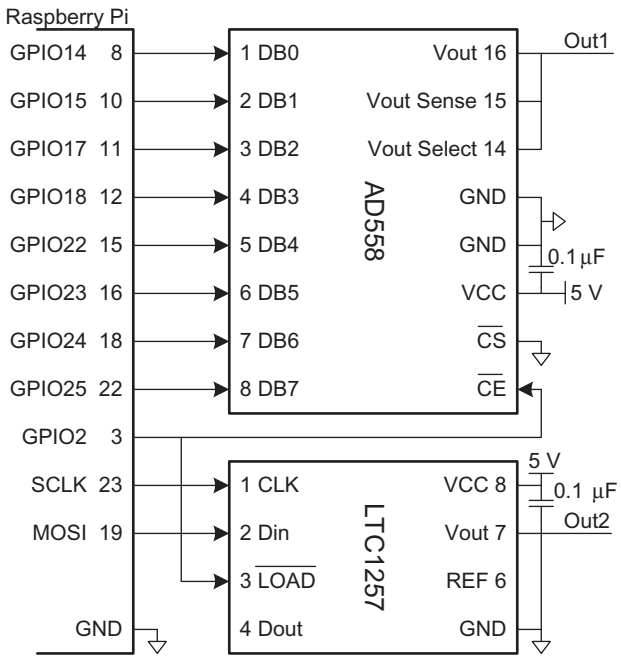


Figure e9.19 DAC parallel and serial interfaces to a Raspberry Pi

to the Pi via SPI0. Both ADCs use a 5 V power supply and have a 0.1 µF decoupling capacitor to reduce power supply noise. The active-low chip enable and load signals on the DACs indicate when to convert the next digital input. They are driven high while a new input is being loaded.

The program is listed below. `pinsMode` and `digitalWrites` are similar to `pinMode` and `digitalWrite` but operate on an array of pins. The program sets the 8 parallel port pins to be outputs and also configures GPIO2 as an output to drive the chip enable and load signals. It initializes the SPI to 1.4 MHz. `initWaveTables` precomputes an array of sample values for the sine and triangle waves. The sine wave is set to a 12-bit scale and the triangle to an 8-bit scale. There are 64 points per period of each wave; changing this value trades precision for frequency. `genWaves` cycles through the samples. It disables interrupts to avoid switching processes and garbling the waves. For each sample, it disables the CE and LOAD signals to the DACs, sends the new sample over the parallel and serial ports, reenables the DACs, and then waits until the timer indicates that it is time for the next sample. `spiSendReceive16` transmits two bytes, but the LTC1257 only cares about the last 12 bits sent. The maximum frequency of somewhat more than 1000 Hz (64 Ksamples/sec) is set by the time to send each point in the `genWaves` function, of which the SPI transmission is a major component.

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

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];
int parallelPins[8] = {14,15,17,18,22,23,24,25};

void initWaveTables(void) {
    int i;
    for (i=0; i<NUMPTS; i++) {
        sine[i] = 2047*(sin(2*3.14159*i/NUMPTS) + 1);  // 12-bit scale
        if (i<NUMPTS/2) triangle[i] = i*511/NUMPTS;    //  8-bit scale
        else triangle[i] = 510-i*511/NUMPTS;
    }
}

void genWaves(int freq) {
    int i, j;
    int microPeriod = 1000000/(NUMPTS*freq);

    noInterrupts();     // disable interrupts to get accurate timing
    for (i=0; i<2000; i++){
        for (j=0; j<NUMPTS; j++) {
            SYS_TIMER_C1 = SYS_TIMER_CLO + microPeriod;  // Set time between samples
            SYS_TIMER_CSbits.M1 = 1;                     // Clear timer match
            digitalWrite(2,1);                           // No load while changing inputs
            spiSendReceive16(sine[j]);
            digitalWrites(parallelPins, 8, triangle[j]);
            digitalWrite(2,0);                           // Load new points into DACs
            while (!SYS_TIMER_CSbits.M1);                // Wait until timer matches
        }
```

```
    }
    interrupts();
}

void main(void) {
    pioInit();

    pinsMode(parallelPins, 8, OUTPUT);  // Set pins connected to the AD558 as outputs
    pinMode(2, OUTPUT);                 // Make pin 2 an output to control LOAD and CE
    spiInit(1400000, 0);                // 1.4MHz SPI clock, default settings
    initWaveTables();
    genWaves(1000);
}
```

9.3.6.2 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, as shown in Figure e9.20. The average value of the output is 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.

The BCM2835 has a PWM controller capable of producing two simultaneous outputs. PWM0 is available at GPIO18 as pin function ALT5, while both PWM outputs are available on the stereo audio jack. Figure e9.21 shows the memory map for the PWM unit and for the clock manager that it depends upon.

The PWM_CTL register is used to turn on pulse width modulation. Bit 0 (PWEN1) must be set to enable the output. Bit 7 (MSEN1: mark-space enable) should also be set to produce pulse width modulation of the form of Figure e9.20 in which the output is HIGH for part of the period and LOW for the remainder.

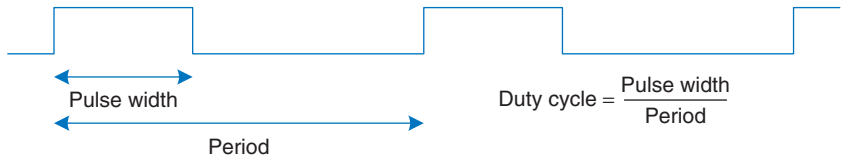


Figure e9.20 Pulse-width modulated (PWM) signal

The PWM signals are derived from a PWM clock generated by the BCM2835 clock manager. The PWM_RNG1 and PWM_DAT1 registers control the period and duty cycle, respectively, by specifying the number of PWM clock ticks for the overall waveform and for the HIGH portion. For example, if the clock manager produces a 25 MHz clock and PWM_RNG1 = 1000 and PWM_DAT1 = 300, the PWM output will operate at (25 MHz / 1000) = 25 kHz and the duty cycle will be 300 / 1000 = 30%.

The clock manager is configured using the CM_PWMCTL and the frequency is set using the CM_PWMDIV register. Table e9.7 summarizes the bit fields of the CM_PWMCTL register. The maximum frequency of the PWM clock is 25 MHz. It can be obtained from the 500 MHz PLLD clock on the Pi as follows:

- CM_PWMCTL: Write 0x5A to PASSWD and 1 to KILL to stop the clock generator
- CM_PWMCLT: Wait for BUSY to clear to indicate the clock is stopped
- CM_PWMCTL: Write 0x5A to PASSWD, 1 to MASH, and 6 to SRC to select PLLD with no audio noise shaping
- CM_PWMDIV: Write 0x5A to PASSWD and 20 to bits 23:12 to divide PLLD by 20 from 500 MHz down to 25 MHz
- CM_PWMCTL: Write 0x5A to PASSWD and 1 to ENAB to restart the clock generator
- CM_PWMCTL: Wait for BUSY to set to indicate the clock is running

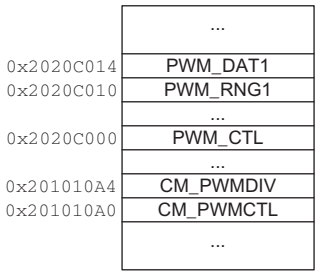


Figure e9.21 PWM and clock manager registers

The CM_PWM registers are not documented in the BCM2835 datasheet. You may find information on them by searching the Internet for "BCM2835 Audio & PWM Clocks" by G.J. van Loo.

Table e9.7 CM_PWMCTL register fields

Bit	Name	Description
31:24	PASSWD	Must be set to 5A when writing
10:9	MASH	Audio noise shaping
7	BUSY	Clock generator running
5	KILL	Write a 1 to stop the clock generator
4	ENAB	Write a 1 to start the clock generator
3:0	SRC	Clock source

Example e9.8 ANALOG OUTPUT WITH PWM

Write an `analogWrite(val)` function to generate an analog output voltage using PWM and an external RC filter. The function should accept an input between 0 (for 0 V output) and 255 (for full 3.3 V output).

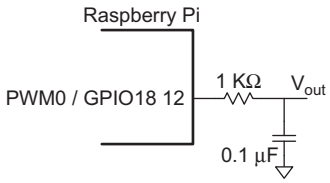


Figure e9.22 Analog output using PWM and low-pass filter

Solution: Use PWM0 to produce a 78.125 kHz signal on GPIO18. The low pass filter in Figure e9.22 has a corner frequency of

$$f_c = \frac{1}{2\pi RC} = 1.6 \text{ kHz}$$

to eliminate the high-speed oscillations and pass the average value.

The PWM functions in EasyPIO are given below. `pwmInit` initializes the PWM module on GPIO18 as described above. `setPWM` sets the frequency and duty cycle of the PWM output. Duty should be between 0 (always OFF) and 1 (always ON). The `analogWrite` function sets the duty cycle based on a full scale of 255.

```
    // Default PLLD value is 500 [MHz]
    #define PLL_FREQUENCY 500000000
    // Max pwm clk is 25 [MHz]
    #define CM_FREQUENCY 25000000
    #define PLL_CLOCK_DIVISOR (PLL_FREQUENCY / CM_FREQUENCY)

void pwmInit() {
    pinMode(18, ALT5);

    // Configure the clock manager to generate a 25 MHz PWM clock.
    // Documentation on the clock manager is missing in the datasheet
    // but found in "BCM2835 Audio and PWM Clocks" by G.J. van Loo 6 Feb 2013.
    // Maximum operating frequency of PWM clock is 25 MHz.
    // Writes to the clock manager registers require simultaneous writing
    // a "password" of 5A to the top bits to reduce the risk of accidental writes.

    CM_PWMCTL = 0; // Turn off PWM before changing
    CM_PWMCTL = PWM_CLK_PASSWORD|0x20; // Turn off clock generator
    while (CM_PWMCTLbits.BUSY); // Wait for generator to stop
    CM_PWMCTL = PWM_CLK_PASSWORD|0x206; // Src = unfiltered 500 MHz CLKD
    CM_PWMDIV = PWM_CLK_PASSWORD|(PLL_CLOCK_DIVISOR << 12); // 25 MHz
    CM_PWMCTL = CM_PWMCTL|PWM_CLK_PASSWORD|0x10;  // Enable PWM clock
    while (!CM_PWMCTLbits.BUSY);  // Wait for generator to start
    PWM_CTLbits.MSEN1 = 1;  // Channel 1 in mark/space mode
    PWM_CTLbits.PWEN1 = 1;  // Enable PWM
}

void setPWM(float freq, float duty) {
    PWM_RNG1 = (int)(CM_FREQUENCY / freq);
    PWM_DAT1 = (int)(duty * (CM_FREQUENCY / freq));
}
```

```
void analogWrite(int val) {
  setPWM(78125, val/255.0);
}
```

The main function tests the PWM by setting the output to half scale (1.65 V).

```
#include "EasyPIO.h"

void main(void) {
  pioInit();
  pwmInit();
  analogWrite(128);
}
```

9.3.6.3 A/D Conversion

The BCM2835 has no built-in ADC, so this section describes A/D conversion using an external converter similar to the external DAC.

Example e9.9 ANALOG INPUT WITH AN EXTERNAL ADC

Interface a 10-bit MCP3002 A/D converter to a Raspberry Pi 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 e9.23 shows a schematic of the connection. The MCP3002 uses VDD as its full scale reference. It accepts a 3.3–5.5 V supply and we

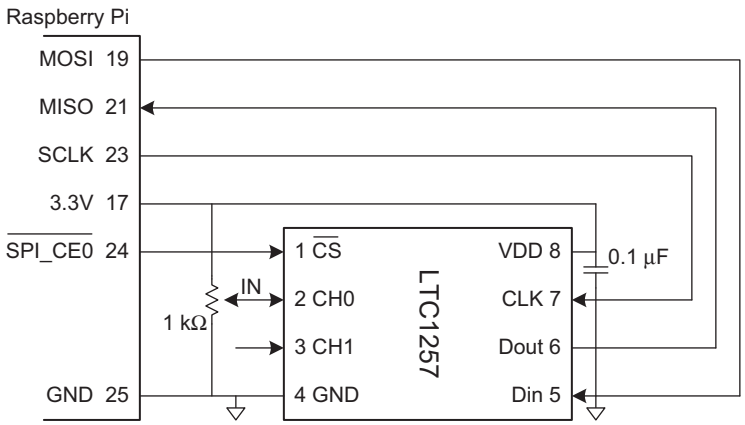


Figure e9.23 **Analog input using external ADC**

choose 3.3 V. The ADC has two input channels, and we connect channel 0 to a potentiometer that we can rotate to adjust the input voltage between 0 and 3.3 V.

The Pi code initializes the SPI and repeatedly reads and prints samples. According to the datasheet, the Raspberry Pi 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. The converter also requires a chip select signal, conveniently provided by the SPI chip enable.

```
#include "EasyPIO.h"

void main(void) {
    int sample;

    pioInit();
    spiInit(200000, 0); // 200 kHz SPI clock, default settings

    while (1){
        sample = spiSendReceive16(0x6000);
        printf("Read %d\n", sample);
    }
}
```

9.3.7 Interrupts

So far, we have relied on *polling*, in which the program continually checks 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 can stop regular program execution and jump to an interrupt handler that responds to the interrupt, then return seamlessly to where it left off.

The Raspberry Pi normally runs Linux, which intercepts interrupts before they get to the program. Therefore, it is presently not straightforward to write interrupt-based programs and this text does not provide examples on the Pi.

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 control. Standard communication interfaces including USB and Ethernet are described in Sections 9.6.1 and 9.6.4.

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 gives an example of interfacing a Raspberry Pi to a character LCD over an 8-bit parallel interface. The interface is compatible with the industry-standard HD44780 LCD controller originally developed by Hitachi. Figure e9.24 shows a Crystalfontz CFAH2002A-TMI-JT 20 × 2 parallel LCD.

Figure e9.25 shows the LCD connected to a Pi over an 8-bit parallel interface. The logic operates at 5 V but is compatible with 3.3 V inputs from the Pi. The LCD contrast is set by a second voltage produced with a potentiometer; it is usually most readable at a setting of 4.2–4.8 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 byte is ready). When the instruction is read, bit 7 returns the busy flag, indicating 1 when busy and 0 when the LCD is ready to accept another instruction.

To initialize the LCD, the Pi must write a sequence of instructions to the LCD as given in Table e9.8. The instructions are written by holding RS = 0 and R/$\overline{W}$ = 0, putting the value on the eight data lines, and pulsing E. After each instruction, it must wait for at least a specified amount of time (or sometimes until the busy flag is clear).

Figure e9.24 Crystalfontz CFAH2002A-TMI 20×2 character LCD (© 2012 Crystalfontz America; reprinted with permission.)

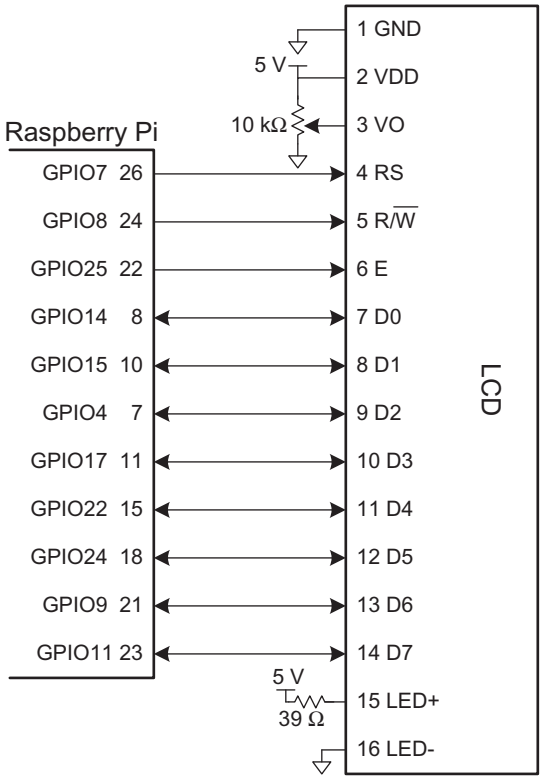


Figure e9.25 Parallel LCD interface

Table e9.8 LCD initialization sequence

Write	Purpose	Wait (μs)
(apply V_{DD})	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	Set 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 to increment cursor after each character	Until busy flag is clear
0x0C	Turn display ON with no cursor	

Then, to write text to the LCD, the Pi can send a sequence of ASCII characters. After each character, it must wait for the busy bit to clear. 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 a character display.

Solution: The following program writes "I love LCDs" to the display by initializing the display and then sending the characters.

```
#include "EasyPIO.h"

int LCD_IO_Pins[] = {14, 15, 4, 17, 22, 24, 9, 11};

typedef enum {INSTR, DATA} mode;
#define RS 7
#define RW 8
#define E  25

char lcdRead(mode md) {
    char c;
    pinsMode(LCD_IO_Pins, 8, INPUT);
    digitalWrite(RS,(md == DATA));      // Set instr/data mode
    digitalWrite(RW, 1);                // Read mode
    digitalWrite(E, 1);                 // Pulse enable
    delayMicros(10);                    // Wait for LCD response
    c = digitalReads(LCD_IO_Pins, 8);   // Read a byte from parallel port
    digitalWrite(E, 0);                 // Turn off enable
    delayMicros(10);
    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  (aka: 0)
    digitalWrites(LCD_IO_Pins, 8, val); // Write the char to the parallel port
    digitalWrite(E, 1); delayMicros(10); // Pulse E
    digitalWrite(E, 0); delayMicros(10);
}

void lcdClear(void) {
    lcdWrite(0x01, INSTR); delayMicros(1530);
}
```

```
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:
    delayMicros(15000);
    lcdWrite(0x30, INSTR); delayMicros(4100);
    lcdWrite(0x30, INSTR); delayMicros(100);
    lcdWrite(0x30, INSTR); lcdBusyWait();
    lcdWrite(0x3C, INSTR); lcdBusyWait();
    lcdWrite(0x08, INSTR); lcdBusyWait();
    lcdClear();
    lcdWrite(0x06, INSTR); lcdBusyWait();
    lcdWrite(0x0C, INSTR); lcdBusyWait();
}

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

9.4.2 VGA Monitor

A more flexible display option is to drive a computer monitor. The Raspberry Pi has built-in support for HDMI and Composite video output. This section explains the low-level details of driving a VGA monitor directly from an FPGA.

The *Video Graphics Array* (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 cathode ray tube, 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 scanline, the gun must turn off for a *horizontal blanking interval* to return to the beginning of the next line. After all of the scanlines are complete, the gun must turn off again for a *vertical blanking interval* to return to the upper left corner. The process repeats about 60–75 times per second to refresh the fluorescence and give the visual illusion of a steady image. A liquid crystal display doesn't require the same electron scan gun, but uses the same VGA interface timing for compatibility.

In a 640×480 pixel VGA monitor refreshed at 59.94 Hz, the pixel clock operates at 25.175 MHz, so each pixel is 39.72 ns wide. The full screen can be viewed as 525 horizontal scanlines of 800 pixels each, but only 480 of the scanlines and 640 pixels per scan line actually convey the image, while the remainder are black. A scanline begins with a *back porch*, the blank section on the left edge of the screen. It then contains 640 pixels, followed by a blank *front porch* at the right edge of the screen and a horizontal sync (hsync) pulse to rapidly move the gun back to the left edge.

Figure e9.26(a) shows the timing of each of these portions of the scanline, beginning with the active pixels. The entire scan line is 31.778 μs long. In the vertical direction, the screen starts with a back porch at the top, followed by 480 active scan lines, followed by a front porch at the bottom and a vertical sync (vsync) pulse to return to the top to start the next frame. A new frame is drawn 60 times per second.

Figure e9.26(b) shows the vertical timing; note that the time units are now scan lines rather than pixel clocks. Higher resolutions use a faster pixel clock, up to 388 MHz at 2048×1536 at 85 Hz. For example, 1024×768 at 60 Hz can be achieved with a 65 MHz pixel clock.

The horizontal timing involves a front porch of 16 clocks, hsync pulse of 96 clocks, and back porch of 48 clocks. The vertical timing involves a front porch of 11 scan lines, vsync pulse of 2 lines, and back porch of 32 lines.

Figure e9.27 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 0–0.7 V, with more positive indicating brighter. The voltages should be 0 during the front and

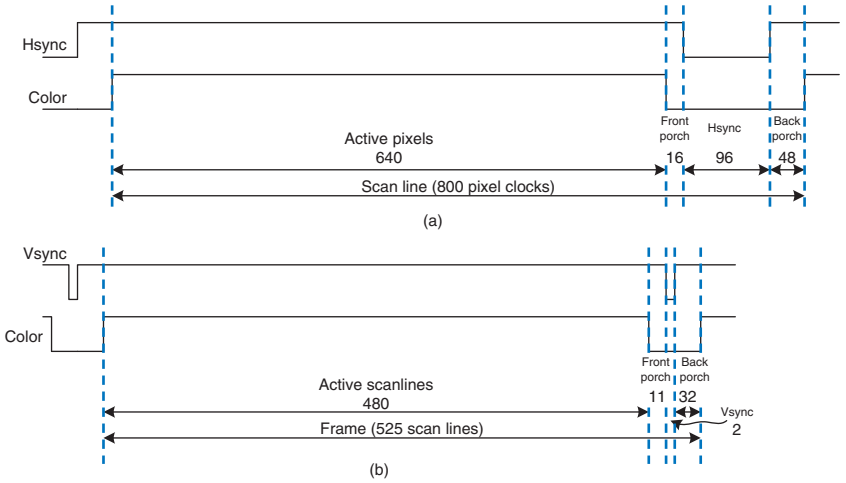


Figure e9.26 VGA timing: (a) horizontal, (b) vertical

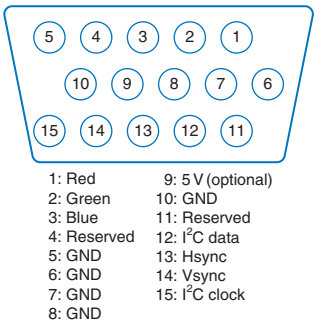


Figure e9.27 VGA connector pinout

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 0.7 V 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 e9.28 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 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.

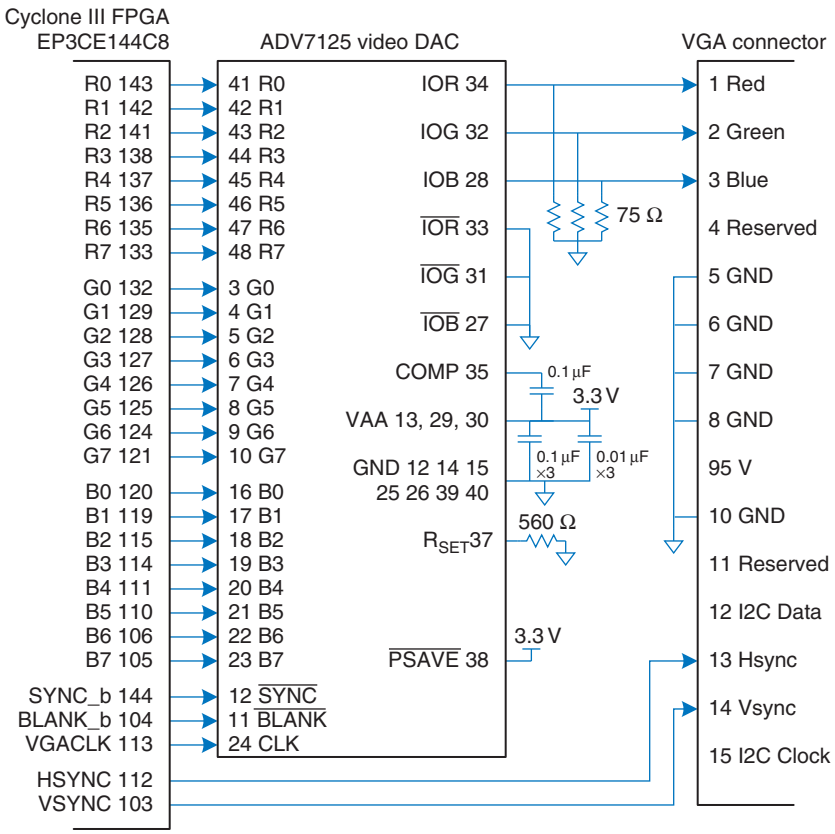


Figure e9.28 **FPGA driving VGA cable through video DAC**

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 e9.28.

Solution: The code assumes a system clock frequency of 40 MHz and uses a phase-locked loop (PLL) on the FPGA to generate the 25.175 MHz VGA 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 e9.29 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.

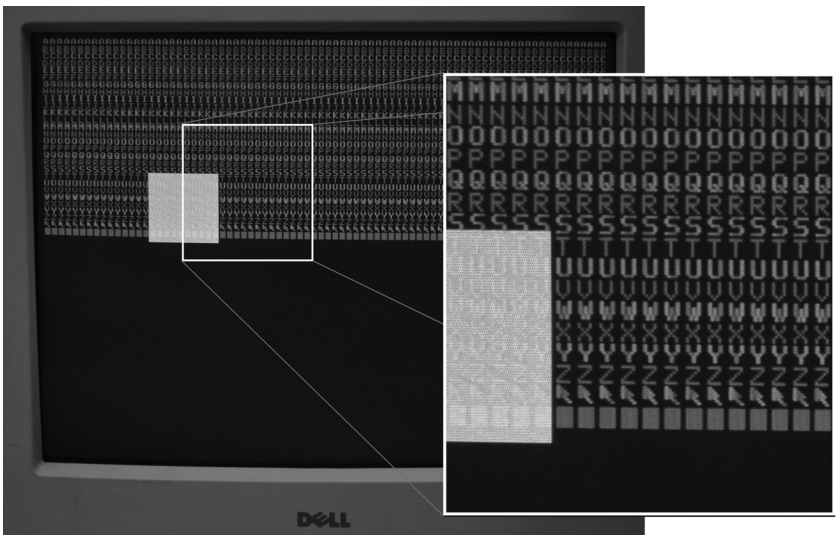


Figure e9.29 VGA output

vga.sv

```
module vga(input  logic       clk,
           output logic       vgaclk,          // 25.175 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;

  // Use a PLL to create the 25.175 MHz VGA pixel clock
  // 25.175 MHz clk period = 39.772 ns
  // Screen is 800 clocks wide by 525 tall, but only 640 x 480 used
  // HSync = 1/(39.772 ns *800) = 31.470 kHz
  // Vsync = 31.474 kHz / 525 = 59.94 Hz (~60 Hz refresh rate)
  pll vgapll(.inclk0(clk), .c0(vgaclk));

  // Generate monitor timing signals
  vgaController vgaCont(vgaclk, 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 HACTIVE = 10'd640,
                                 HFP     = 10'd16,
                                 HSYN    = 10'd96,
                                 HBP     = 10'd48,
                                 HMAX    = HACTIVE + HFP + HSYN + HBP,
                                 VBP     = 10'd32,
                                 VACTIVE = 10'd480,
                                 VFP     = 10'd11,
                                 VSYN    = 10'd2,
                                 VMAX    = VACTIVE + VFP + VSYN + VBP)
                       (input  logic       vgaclk,
                        output logic       hsync, vsync, sync_b, blank_b,
                        output logic [9:0] x, y);
  // counters for horizontal and vertical positions
  always @(posedge vgaclk) begin
    x++;
    if (x == HMAX) begin
      x = 0;
      y++;
      if (y == VMAX) y = 0;
    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 = hsync & vsync;

  // Force outputs to black when outside the legal display area
  assign blank_b = (hcnt < HACTIVE) & (vcnt < VACTIVE);
endmodule

module videoGen(inputlogic [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
```

charrom.txt

```
// 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
...
```

Bluetooth 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 e9.9 Bluetooth classes

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

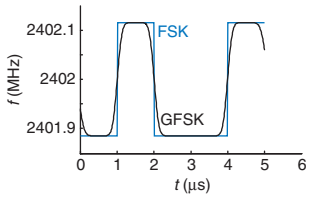


Figure e9.30 FSK and GFSK waveforms

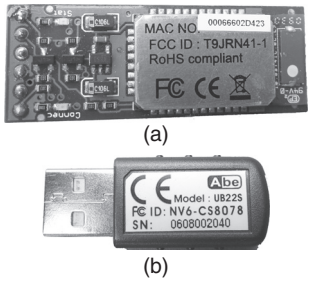


Figure e9.31 BlueSMiRF module and USB dongle

9.4.3 Bluetooth Wireless Communication

There are many standards 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 5–100 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 e9.9, 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 e9.30 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 e9.31(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

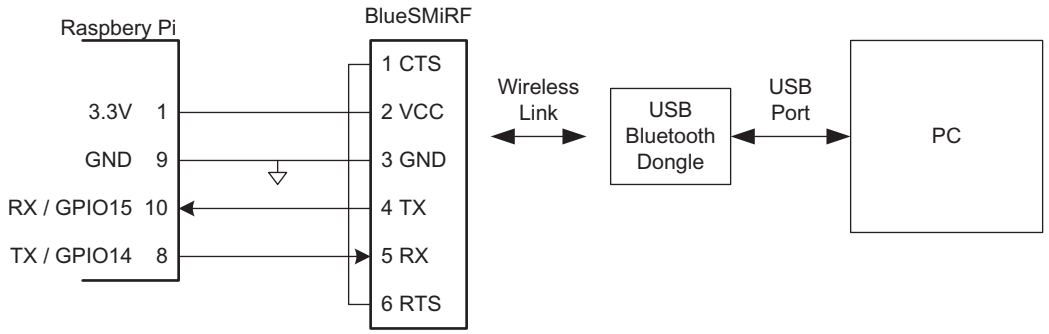


Figure e9.32 **BlueSMiRF Raspberry Pi to PC link**

such as a Bluetooth USB dongle connected to a PC. Thus, it can provide a wireless serial link between a Pi and a PC similar to the link from Figure e9.15 but without the cable. The wireless link is compatible with the same software as is the wired link.

Figure e9.32 shows a schematic for such a link. The TX pin of the BlueSMiRF connects to the RX pin of the Pi, and vice versa. The RTS and CTS pins are connected so that the BlueSMiRF shakes its own hand.

The BlueSMiRF defaults to 115.2 k baud with 8 data bits, 1 stop bit, and no parity or flow control. It operates at 3.3 V digital logic levels, so no RS-232 transceiver is necessary to connect with another 3.3 V device.

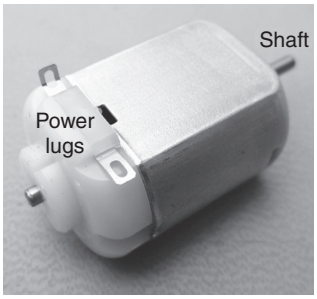
To use the interface, plug a USB Bluetooth dongle into a PC. Power up the Pi and BlueSMiRF. The red STAT light will flash on the BlueSMiRF indicating that it is waiting to make a connection. Open the Bluetooth icon in the PC system tray and use the Add Bluetooth Device Wizard to pair the dongle with 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

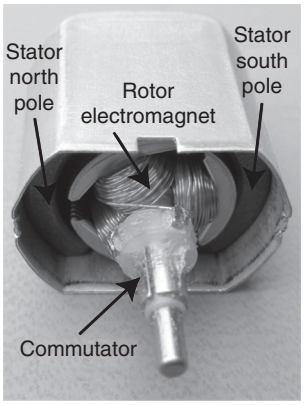
Another major application of microcontrollers is to drive actuators such as motors. This section describes three types of motors: DC motors, servo motors, and stepper motors. *DC motors* require a high drive current, so a powerful driver such as an *H-bridge* must be connected between the microcontroller and the motor. They 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.

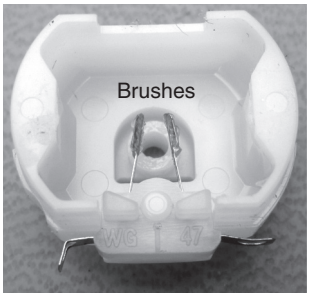
Motors can draw a substantial amount of current and may introduce glitches on the power supply that disturb digital logic. One way to reduce this problem is to use a different power supply or battery for the motor than for the digital logic.



(a)



(b)



(c)

Figure e9.33 **DC motor**

9.4.4.1 DC Motors

Figure e9.33 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 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 e9.34(a). 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. The H-bridge also contains some digital logic to conveniently control the switches.

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 e9.34(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.

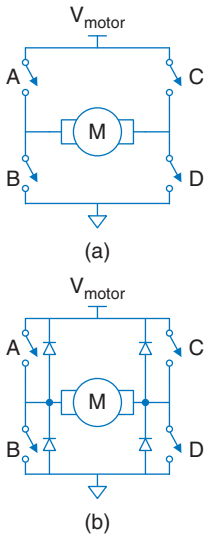


Figure e9.34 H-bridge

Example e9.12 AUTONOMOUS VEHICLE

Design a system in which a Raspberry Pi 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 control the speed of the motors.

Solution: Figure e9.35 shows a pair of DC motors controlled by a Pi via a Texas Instruments SN754410 dual H-bridge. The H-bridge requires a 5 V logic supply V_{CC1} and a 4.5–36 V motor supply V_{CC2}; it has $V_{IH} = 2$ V and is hence compatible with the 3.3 V I/O from the Pi. 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 Pi cannot supply enough current to drive most motors and the Pi could be damaged.

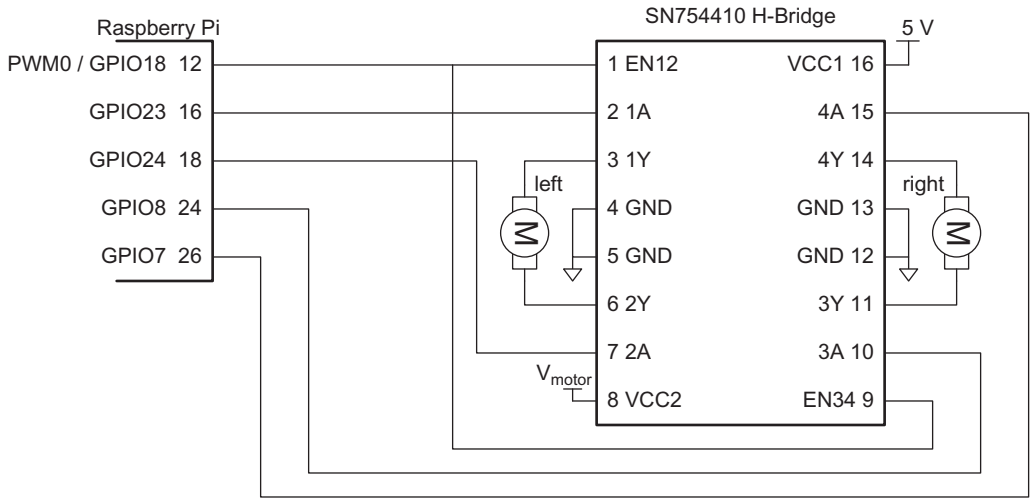


Figure e9.35 Motor control with dual H-bridge

Table e9.10 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

Table e9.10 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.

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.

```
#include "EasyPIO.h"

// Motor Constants
#define MOTOR_1A 23
#define MOTOR_2A 24
#define MOTOR_3A 8
#define MOTOR_4A 7

void setSpeed(float dutycycle) {        // pwmInit() must be called first.
    setPWM(5000, dutycycle);
}

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();                         // Turn on PWM
    setSpeed(0.75);                    // Default to partial power
}

main(void) {
    pioInit();
    initMotors();
    forward(); delayMillis(5000);
    backward(); delayMillis(5000);
    left(); delayMillis(5000);
    right(); delayMillis(5000);
    halt();
}
```

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 e9.36(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 e9.36(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.

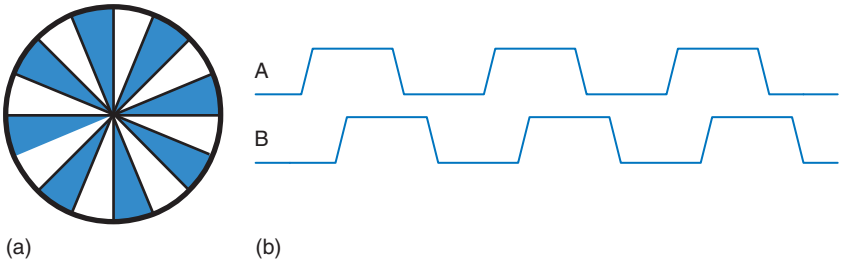


Figure e9.36 Shaft encoder (a) disk, (b) quadrature outputs

9.4.4.2 Servo Motor

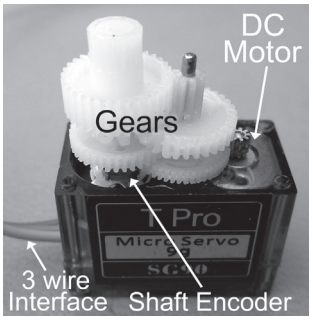


Figure e9.37 SG90 servo motor

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. They have a limited rotation, typically 180°. Figure e9.37 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.

In a typical servo motor with 180 degrees of rotation, a pulse width of 0.5 ms drives the shaft to 0°, 1.5 ms to 90°, and 2.5 ms to 180°. For example, Figure e9.38 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 be difficult. The center pin with a red wire is normally power, and the black or brown wire is normally ground.

Example e9.13 SERVO MOTOR

Design a system in which a Raspberry Pi drives a servo motor to a desired angle.

Solution: Figure e9.39 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 4.0–7.2 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 Raspberry Pi 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

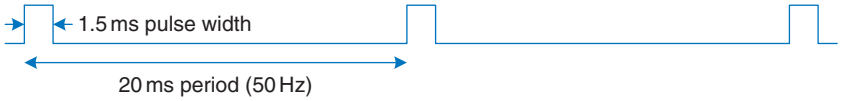


Figure e9.38 Servo control waveform

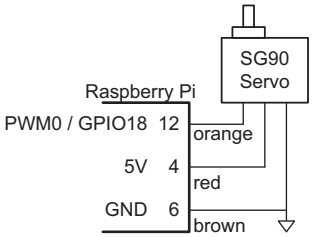


Figure e9.39 Servo motor control

the appropriate duty cycle for the desired angle. It cycles through positioning the servo at 0, 90, and 180 degrees.

```
#include "EasyPIO.h"
void setServo(float angle) {
  setPWM(50.0, 0.025 + (0.1 * (angle / 180)));
}

void main(void) {
  pioInit();
  pwmInit();
  while (1) {
    setServo(0.0);     // Left
    delayMillis(1000);
    setServo(90.0);    // Center
    delayMillis(1000);
    setServo(180.0);   // Right
    delayMillis(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, 2.5 ms indicating full speed forward, and 0.5 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.

9.4.4.3 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 e9.40(a) shows a simplified two-phase bipolar motor with a 90° 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

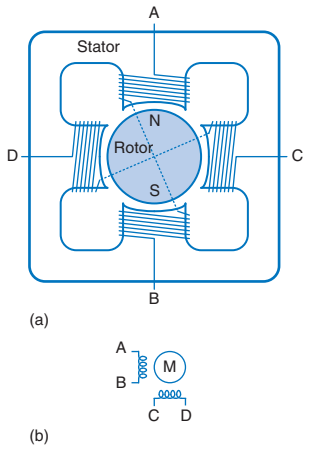


Figure e9.40 **Two-phase bipolar motor: (a) simplified diagram, (b) symbol**

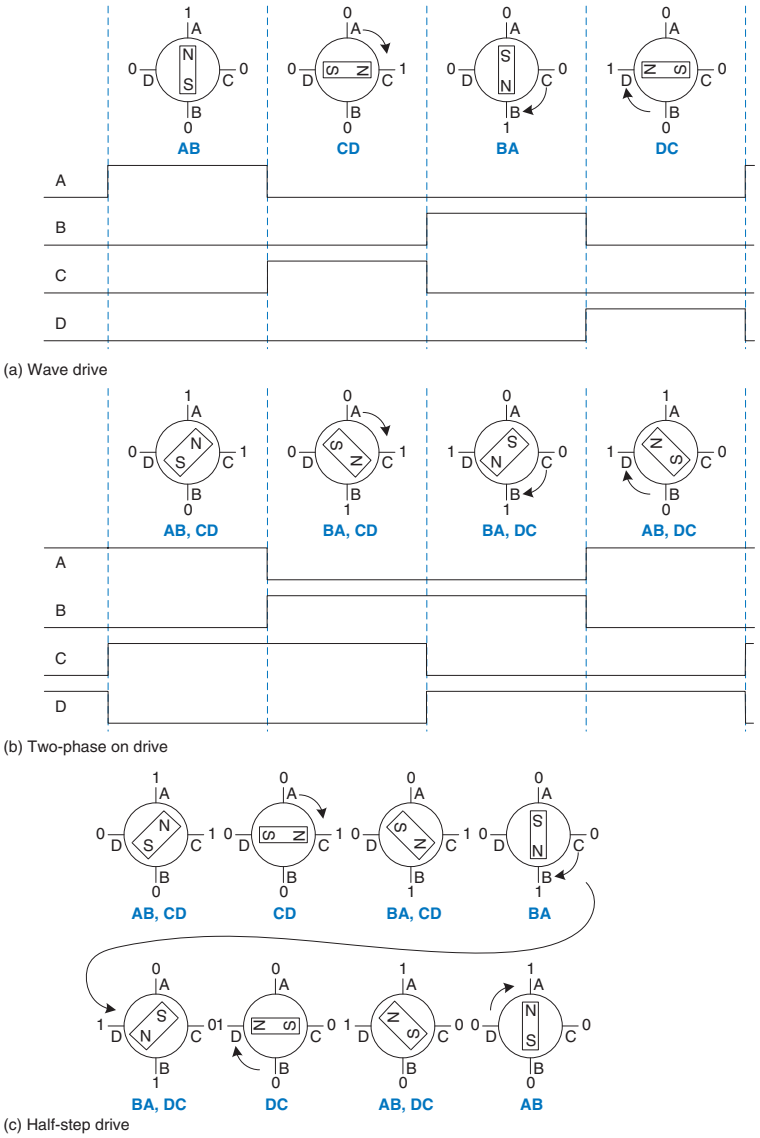


Figure e9.41 **Bipolar motor drive**

the two phases. Two-phase bipolar motors thus have four terminals. Figure e9.40(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 e9.41 shows three common drive sequences for a two phase bipolar motor. Figure e9.41(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 e9.41(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 e9.41(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.

In a real motor, the rotor has many poles to make the angle between steps much smaller. For example, Figure e9.42 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.

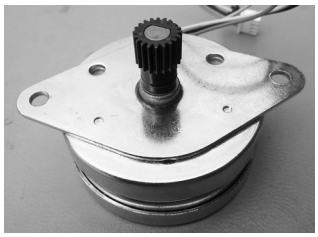


Figure e9.42 AIRPAX LB82773-M1 bipolar stepper motor

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 e9.43(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 e9.43(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 (PWM) 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 shown in Figure e9.43(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.

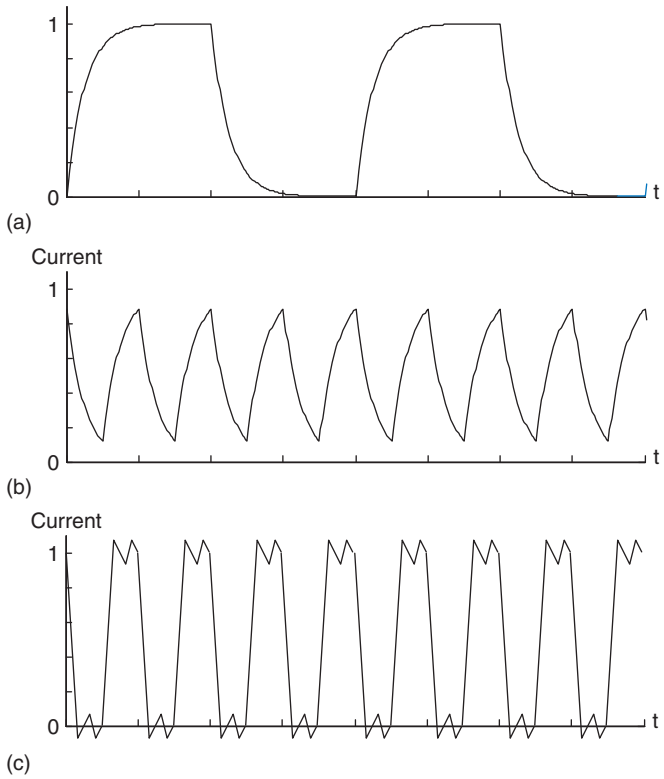


Figure e9.43 **Bipolar stepper motor direct drive current: (a) slow rotation, (b) fast rotation, (c) fast rotation with chopper drive**

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 e9.44 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 else the motor may skip steps as the rotation rate increases.

```
#include "EasyPIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MICROS_PER_SEC 1000000
#define DEG_PER_REV 360

int stepperPins[] = {18, 8, 7, 23, 24};
```

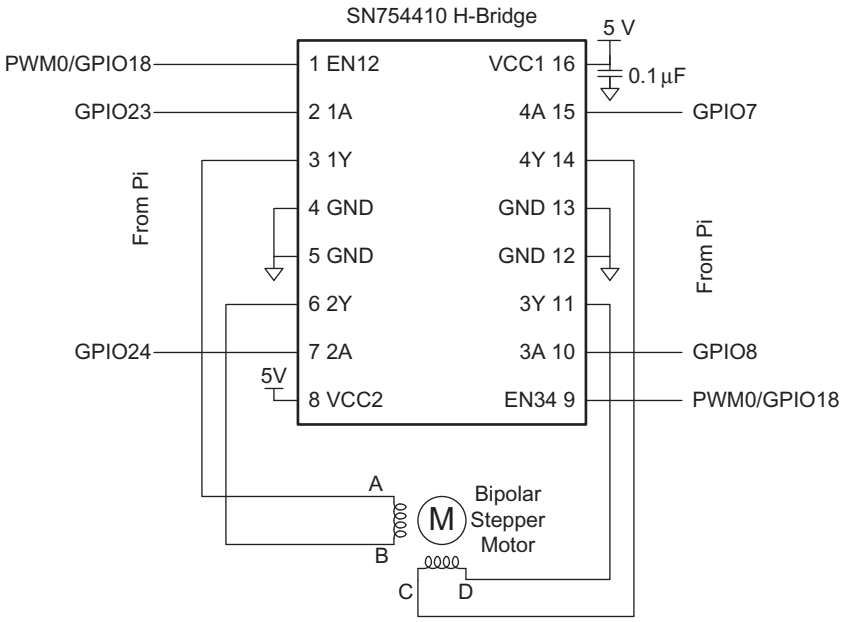


Figure e9.44 **Bipolar stepper motor direct drive with H-bridge**

```
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 microsPerStep = (SECS_PER_MIN * MICROS_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;
        delayMicros(microsPerStep);
    }
}

void main(void) {
    pioInit();
    stepperInit();
    stepperSpin(1, 12000, 120);  // Spin 60 revolutions at 120 rpm
}
```

9.5 BUS INTERFACES

A *bus interface* connects processors to memory and/or peripherals. In general, a bus interface supports one or more *bus masters* that can initiate read or write requests to the bus and one or more *slaves* that respond to the requests; processors are normally masters and memory and peripherals are slaves.

The *Advanced Microcontroller Bus Architecture* (AMBA) is an open standard bus interface for connecting components on a chip. Introduced by ARM in 1996, it has developed through multiple revisions to boost performance and features and has become a *de facto* standard for embedded microcontrollers. The *Advanced High-performance Bus* (AHB) is one of the AMBA standards. AHB-Lite is a simplified version of AHB that supports a single bus master. This section describes AHB-Lite to illustrate the characteristics of a typical bus interface and to show how to design memory and peripherals that interface to a standard bus.

AHB is an example of a point-to-point read bus, in contrast with older bus architectures that use a single shared data bus where each slave accesses the bus via a tristate driver. Using point-to-point links between each slave and the read multiplexer allows the bus to run faster and avoids wasting power when one slave turns on its driver before another has turned off.

9.5.1 AHB-Lite

Figure e9.45 shows a simple AHB-Lite bus connecting a processor (bus master) to RAM, ROM, and two peripherals (slaves). Observe that the bus is very similar to the one from Figure e9.1 except that the names have changed. The master provides a synchronous clock (HCLK) to all of the slaves and can reset the slaves by asserting HRESETn low. The master sends an address. The address decoder uses the most significant bits to generate the HSEL signal selecting which slave to access, and the slaves use the least significant bits to define the memory location or register. The master sends HWDATA for writes. Each slave reads onto its own HRDATA, and a multiplexer chooses the data from the selected slave.

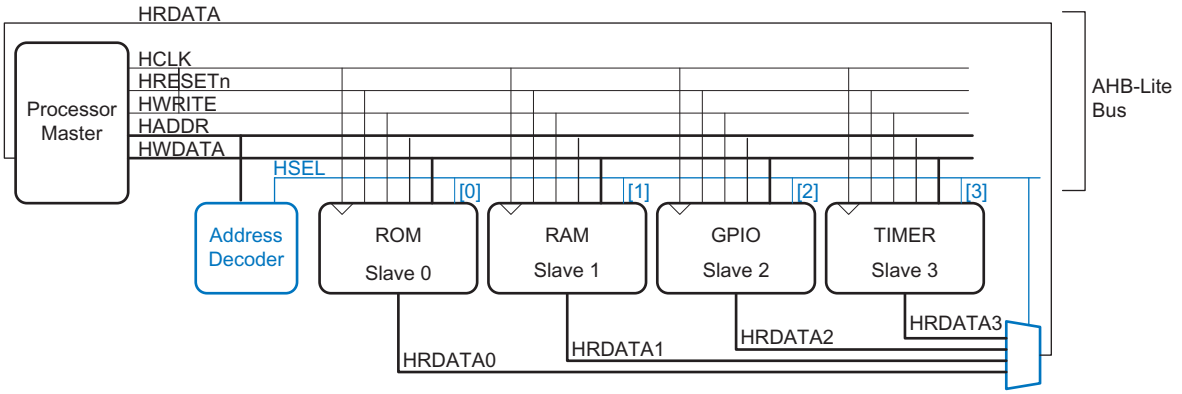


Figure e9.45 AHB-Lite bus

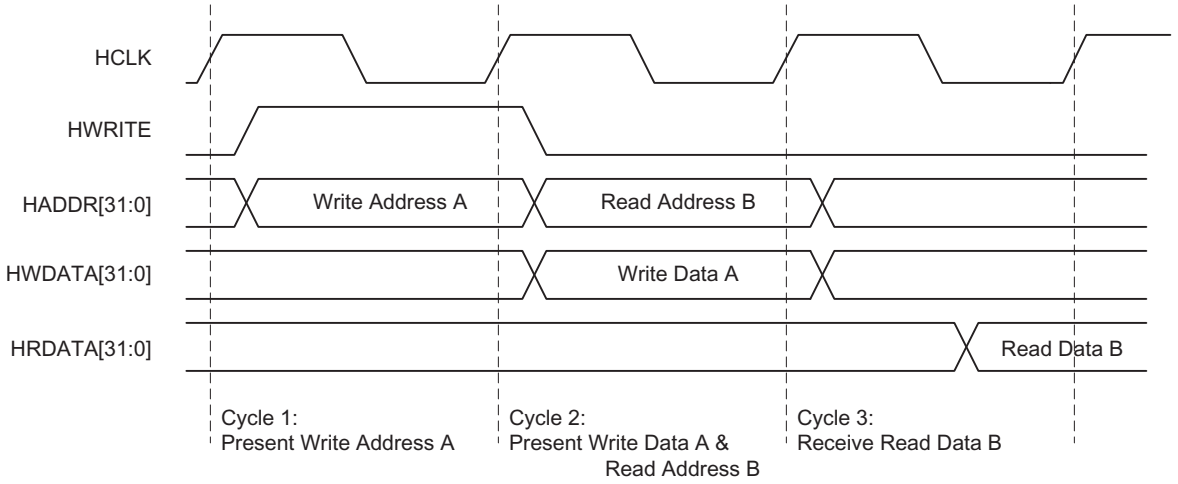


Figure e9.46 **AHB-Lite transfer timing**

The master sends a 32-bit address on one cycle and writes or reads data on the subsequent cycle. The write or read is called a *transfer*. For writes, the master raises HWRITE and sends the 32-bit HWDATA to write. For reads, the master lowers HWRITE and the slave responds with 32-bit HRDATA. Transfers can overlap so that the master can send the address of the next transfer while reading or writing data for the current transfer. Figure e9.46 illustrates the timing of the bus for a write followed immediately by a read. Observe how the data lags one cycle behind the address and how the two transfers partially overlap.

In this example, we assume the bus transfers a single 32-bit word at a time and that the slave responds in one clock cycle. AHB-Lite defines additional signals to specify the size of the transfer (8 – 1024 bits) and to transfer bursts of 4 to 16 elements. The master can also specify types of transfers, protection, and bus locking. Slaves can deassert HREADY to indicate that they need multiple clock cycles to respond, or can assert HRESP to indicate an error. Interested readers should consult the *AMBA 3 AHB-Lite Protocol Specification*, available online.

9.5.2 Memory and Peripheral Interface Example

This section illustrates connecting RAM, ROM, GPIO, and a timer to a processor over an AHB-Lite bus. Figure e9.47 shows a memory map for the system from Figure e9.45 with 128 KB of RAM and 64 KB of ROM. The GPIO controls 32 I/O pins. The 32-bit GPIO_DIR register controls whether each pin is an output (1) or an input (0). The 32-bit

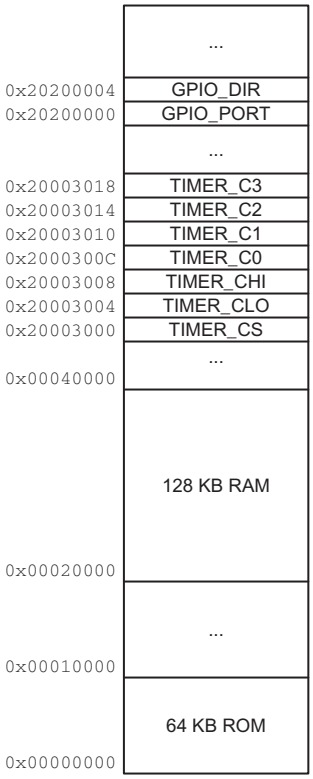


Figure e9.47 **System memory map**

GPIO_PORT register can be written to specify the value of outputs and read to return the values on the pins. The Timer module resembles the BCM2835 counter described in Section 9.3.5, containing a 64-bit counter running at the HCLK frequency (TIMER_CHI:TIMER_CLO), four 32-bit compare channels (TIMER_C3:0), and a match register (TIMER_CS).

HDL Example e9.1 lists SystemVerilog code for the system. The decoder is based on the memory map. The memories and peripherals interface to the bus. Unnecessary signals are omitted; for example, the ROM ignores writes. The GPIO module also connects to 32 I/O pins that can behave as inputs or outputs.

HDL Example e9.1

```
module ahb_lite(input logic          HCLK,
                input logic          HRESETn,
                input logic  [31:0]  HADDR,
                input logic          HWRITE,
                input logic  [31:0]  HWDATA,
                output logic [31:0]  HRDATA,
                inout tri    [31:0]  pins);

  logic [3:0] HSEL;
  logic [31:0] HRDATA0, HRDATA1, HRDATA2, HRDATA3;
  logic [31:0] pins_dir, pins_out, pins_in;
  logic [31:0] HADDRDEL;
  logic HWRITEDEL;

  // Delay address and write signals to align in time with data
  flop #(32) adrreg(HCLK, HADDR, HADDRDEL);
  flop #(1)       writereg(HCLK, HWRITE, HWRITEDEL);

  // Memory map decoding
  ahb_decoder dec(HADDRDEL, HSEL);
  ahb_mux  mux(HSEL, HRDATA0, HRDATA1, HRDATA2, HRDATA3,
               HRDATA);

  // Memory and peripherals
  ahb_rom     rom (HCLK, HSEL[0], HADDRDEL[15:2], HRDATA0);
  ahb_ram     ram (HCLK, HSEL[1], HADDRDEL[16:2], HWRITEDEL,
              HWDATA, HRDATA1);
  ahb_gpio    gpio (HCLK, HRESETn, HSEL[2], HADDRDEL[2],
                    HWRITEDEL, HWDATA, HRDATA2, pins);
  ahb_timer   timer(HCLK, HRESETn, HSEL[3], HADDRDEL[4:2],
                    HWRITEDEL, HWDATA, HRDATA3);
endmodule

module ahb_decoder(input logic [31:0] HADDR,
                   output logic [3:0] HSEL);

  // Decode based on most significant bits of the address
  assign HSEL[0]=(HADDR[31:16]
                  ==16'h0000); // 64KB ROM at 0x00000000 -
                               0x0000FFFF
  assign HSEL[1]=(HADDR[31:17]
                  ==15'h0001); // 128KB RAM at 0x00020000 -
                               0x003FFFFF
  assign HSEL[2]=(HADDR[31:4]
                  ==28'h2020000); // GPIO at 0x20200000 -
                  0x20200007
assign HSEL[3]=(HADDR[31:8]
                ==24'h200030); // Timer at 0x20003000 -
                               0x2000301B
endmodule

module ahb_mux(input logic [3:0] HSEL,
               input logic [31:0] HRDATA0, HRDATA1, HRDATA2,
               HRDATA3,
               output logic [31:0] HRDATA);

  always_comb
    casez(HSEL)
      4'b???1: HRDATA <= HRDATA0;
      4'b??10: HRDATA <= HRDATA1;
      4'b?100: HRDATA <= HRDATA2;
      4'b1000: HRDATA <= HRDATA3;
    endcase
endmodule

module ahb_ram(input logic        HCLK,
               input logic        HSEL,
               input logic [16:2] HADDR,
               input logic        HWRITE,
               input logic [31:0] HWDATA,
               output logic [31:0] HRDATA);

  logic [31:0] ram[32767:0]; // 128KB RAM organized as 32K
                             x 32 bits

  assign HRDATA = ram[HADDR]; // *** check addressing is
                              correct
  always_ff @(posedge HCLK)
    if (HWRITE & HSEL) ram[HADDR] <= HWDATA;
endmodule

module ahb_rom(input logic        HCLK,
               input logic        HSEL,
               input logic [16:2] HADDR,
               output logic [31:0] HRDATA);
  logic [31:0] rom[16383:0]; // 64KB ROM organized as 16K x
                             32 bits

  // *** load ROM from disk file

  assign HRDATA = rom[HADDR]; // *** check addressing is
                              correct

endmodule
```

```
module ahb_gpio(input  logic        HCLK,
                input  logic        HRESETn,
                input  logic        HSEL,
                input  logic [2]    HADDR,
                input  logic        HWRITE,
                input  logic [31:0] HWDATA,
                output logic [31:0] HRDATA,
                output logic [31:0] pin_dir,
                output logic [31:0] pin_out,
                input  logic [31:0] pin_in);

  logic [31:0] gpio[1:0];  // GPIO registers

  // write selected register
  always_ff @(posedge HCLK or negedge HRESETn)
    if (~HRESETn) begin
      gpio[0] <= 32'b0;  // GPIO_PORT
      gpio[1] <= 32'b0;  // GPIO_DIR
    end else if (HWRITE & HSEL)
      gpio[HADDR] <= HWDATA;

  // read selected register
  assign HRDATA = HADDR ? gpio[1] : pin_in;

  // send value and direction to I/O drivers
  assign pin_out = gpio[0];
  assign pin_dir = gpio[1];
endmodule

module ahb_timer(input  logic        HCLK,
                 input  logic        HRESETn,
                 input  logic        HSEL,
                 input  logic [4:2]  HADDR,
                 input  logic        HWRITE,
                 input  logic [31:0] HWDATA,
                 output logic [31:0] HRDATA);

  logic [31:0] timers[6:0]; // timer registers
  logic [31:0] chi, clo;    // next counter value
  logic [3:0]  match, clr;  // determine if counter matches
                               compare reg

  // write selected register and update tiers and match
    always_ff @(posedge HCLK or negedge HRESETn)
  if (~HRESETn) begin
    timers[0] <= 32'b0;  // TIMER_CS
    timers[1] <= 32'b0;  // TIMER_CLO
    timers[2] <= 32'b0;  // TIMER_CHI
    timers[3] <= 32'b0;  // TIMER_CO
    timers[4] <= 32'b0;  // TIMER_C1
    timers[5] <= 32'b0;  // TIMER_C2
    timers[6] <= 32'b0;  // TIMER_C3
  end else begin
    timers[0] <= {28'b0, match};
    timers[1] <= (HWRITE & HSEL & HADDR == 3'b000) ?
                 HWDATA : clo
    timers[2] <= (HWRITE & HSEL & HADDR == 3'b000) ?
                 HWDATA : chi;
    if (HWRITE & HSEL & HADDR == 3'b011) timers[3] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b100) timers[4] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b101) timers[5] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b110) timers[6] <= HWDATA;
  end

  // read selected register
  assign HRDATA = timers[HADDR];

  // increment 64-bit counter as pair of TIMER_CHI, TIMER_CLO
  assign {chi, clo} = {timers[2], timers[1]} + 1;

  // generate matches: set match bit when counter matches
     compare register
  //  clear bit when a 1 is written to that position of the match
      register
  assign clr = (HWRITE & HSEL & HADDR == 3'b000 & HWDATA[3:0]);
  assign match[0] = ~clr[0] & (timers[0][0] |
                    (timers[1] == timers[3]));
  assign match[1] = ~clr[1] & (timers[0][1] |
                    (timers[1] == timers[4]));
  assign match[2] = ~clr[2] & (timers[0][2] |
                    (timers[1] == timers[5]));
  assign match[3] = ~clr[3] & (timers[0][3] |
                    (timers[1] == timers[6]));
endmodule

module gpio_pins(input  logic [31:0] pin_dir, // 1 = output,
                                                 0 = input
                 input  logic [31:0] pin_out, // value to drive
                                                 on outputs
                 output logic [31:0] pin_in,  // value read
                                                 from pins
                 inout  tri   [31:0] pin);  // tristate pins

  // Individual tristate control of each pin

  // No graceful way to control tristates on a per-bit basis in
     SystemVerilog
  genvar i;
  generate
  for (i=0; i<32; i=i+1) begin: pinloop
    assign pin[i] = pin_dir[i] ? pin_out[i] : 1'bz;
  end
  endgenerate

  assign pin_in = pin;
endmodule
```

9.6 PC I/O SYSTEMS

Personal computers (PCs) use a wide variety of I/O protocols for purposes including memory, disks, networking, internal expansion cards, and external devices. These I/O standards have evolved to offer very high performance and to make it easy for users to add devices. These attributes

come at the expense of complexity in the I/O protocols. This section explores the major I/O standards used in PCs and examines some options for connecting a PC to custom digital logic or other external hardware.

Figure e9.48 shows a PC motherboard for a Core i5 or i7 processor. The processor is packaged in a *land grid array* with 1156 gold-plated pads to supply power and ground to the processor and connect the processor to memory and I/O devices. The motherboard contains the DRAM memory module slots, a wide variety of I/O device connectors, and the power supply connector, voltage regulators, and capacitors. A pair of DRAM modules are connected over a DDR3 interface. External peripherals such as keyboards or webcams are attached over USB. High-performance expansion cards such as graphics cards connect over the PCI Express x16 slot, while lower-performance cards can use PCI Express x1 or the older PCI slots. The PC connects to the network using the Ethernet jack. The hard disk connects to a SATA port. The remainder of this section gives an overview of the operation of each of these I/O standards.

One of the major advances in PC I/O standards has been the development of high-speed serial links. Until recently, most I/O was built around parallel links consisting of a wide data bus and a clock signal. As data rates increased, the difference in delay among the wires in the bus set a limit to how fast the bus could run. Moreover, busses connected to multiple devices suffer from transmission line problems such as reflections and different flight times to different loads. Noise can also corrupt the data. Point-to-point serial

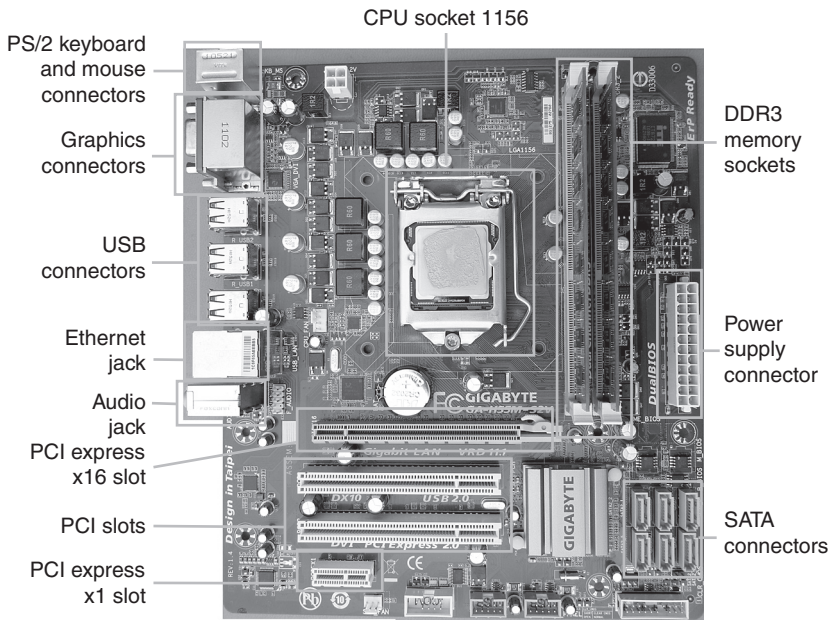


Figure e9.48 **Gigabyte GA-H55M-S2V motherboard**

links eliminate many of these problems. The data is usually transmitted on a differential pair of wires. External noise that affects both wires in the pair equally is unimportant. The transmission lines are easy to properly terminate, so reflections are small (see Section A.8 on transmission lines). No explicit clock is sent; instead, the clock is recovered at the receiver by watching the timing of the data transitions. High-speed serial link design is a specialized subject, but good links can run faster than 10 Gb/s over copper wires and even faster along optical fibers.

9.6.1 USB

Until the mid-1990s, adding a peripheral to a PC took some technical savvy. Adding expansion cards required opening the case, setting jumpers to the correct position, and manually installing a device driver. Adding an RS-232 device required choosing the right cable and properly configuring the baud rate, and data, parity, and stop bits. The *Universal Serial Bus* (USB), developed by Intel, IBM, Microsoft, and others, greatly simplified adding peripherals by standardizing the cables and software configuration process. Billions of USB peripherals are now sold each year.

USB 1.0 was released in 1996. It uses a simple cable with four wires: 5 V, GND, and a differential pair of wires to carry data. The cable is impossible to plug in backward or upside down. It operates at up to 12 Mb/s. A device can pull up to 500 mA from the USB port, so keyboards, mice, and other peripherals can get their power from the port rather than from batteries or a separate power cable.

USB 2.0, released in 2000, upgraded the speed to 480 Mb/s by running the differential wires much faster. With the faster link, USB became practical for attaching webcams and external hard disks. Flash memory sticks with a USB interface also replaced floppy disks as a means of transferring files between computers.

USB 3.0, released in 2008, further boosted the speed to 5 Gb/s. It uses the same shape connector, but the cable has more wires that operate at very high speed. It is well suited to connecting high-performance hard disks. At about the same time, USB added a Battery Charging Specification that boosts the power supplied over the port to speed up charging mobile devices.

The simplicity for the user comes at the expense of a much more complex hardware and software implementation. Building a USB interface from the ground up is a major undertaking. Even writing a simple device driver is moderately complex.

9.6.2 PCI and PCI Express

The *Peripheral Component Interconnect* (PCI) bus is an expansion bus standard developed by Intel that became widespread around 1994. It was

used to add expansion cards such as extra serial or USB ports, network interfaces, sound cards, modems, disk controllers, or video cards. The 32-bit parallel bus operates at 33 MHz, giving a bandwidth of 133 MB/s.

The demand for PCI expansion cards has steadily declined. More standard ports such as Ethernet and SATA are now integrated into the motherboard. Many devices that once required an expansion card can now be connected over a fast USB 2.0 or 3.0 link. And video cards now require far more bandwidth than PCI can supply.

Contemporary motherboards often still have a small number of PCI slots, but fast devices like video cards are now connected via *PCI Express* (PCIe). PCI Express slots provide one or more lanes of high-speed serial links. In PCIe 3.0, each lane operates at up to 8 Gb/s. Most motherboards provide an x16 slot with 16 lanes giving a total of 16 GB/s of bandwidth to data-hungry devices such as video cards.

9.6.3 DDR3 Memory

DRAM connects to the microprocessor over a parallel bus. In 2015, the present standard is DDR3, a third generation of double-data rate memory bus operating at 1.5 V. Typical motherboards now come with two DDR3 channels so they can simultaneously access two banks of memory modules. DDR4 is emerging, operating at 1.2V and higher speed.

Figure e9.49 shows a 4 GB DDR3 dual inline memory module (DIMM). The module has 120 contacts on each side, for a total of 240 connections, including a 64-bit data bus, a 16-bit time-multiplexed address bus, control signals, and numerous power and ground pins. In 2015, DIMMs typically carry 1–16 GB of DRAM. Memory capacity has been doubling approximately every 2–3 years.

DRAM presently operates at a clock rate of 100–266 MHz. DDR3 operates the memory bus at four times the DRAM clock rate. Moreover, it transfers data on both the rising and falling edges of the clock. Hence, it sends 8 words of data for each memory clock. At 64 bits/word, this corresponds to 6.4–17 GB/s of bandwidth. For example, DDR3-1600 uses a 200 MHz memory clock and an 800 MHz I/O clock to send 1600 million words/sec, or 12800 MB/s. Hence, the modules are

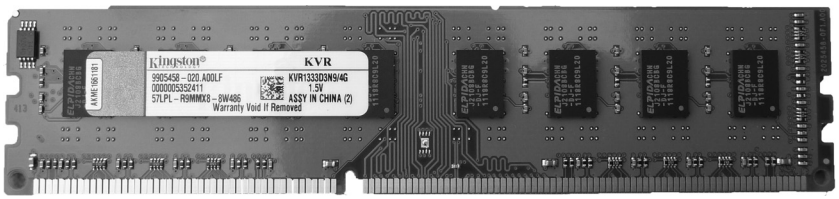


Figure e9.49 DDR3 memory module

also called PC3-12800. Unfortunately, DRAM latency remains high, with a roughly 50 ns lag from a read request until the arrival of the first word of data.

9.6.4 Networking

Computers connect to the Internet over a network interface running the *Transmission Control Protocol and Internet Protocol* (TCP/IP). The physical connection may be an Ethernet cable or a wireless Wi-Fi link.

Ethernet is defined by the IEEE 802.3 standard. It was developed at Xerox Palo Alto Research Center (PARC) in 1974. It originally operated at 10 Mb/s (called 10 Mbit Ethernet), but now is commonly found at 100 Mbit (Mb/s) and 1 Gbit (Gb/s) running on Category 5 cables containing four twisted pairs of wires. 10 Gbit Ethernet running on fiber optic cables is increasingly popular for servers and other high-performance computing, and 100 Gbit Ethernet is emerging.

Wi-Fi is the popular name for the IEEE 802.11 wireless network standard. It operates in the 2.4 and 5 GHz unlicensed wireless bands, meaning that the user doesn't need a radio operator's license to transmit in these bands at low power. Table e9.11 summarizes the capabilities of three generations of Wi-Fi; the emerging 802.11ac standard promises to push wireless data rates beyond 1 Gb/s. The increasing performance comes from advancing modulation and signal processing, multiple antennas, and wider signal bandwidths.

9.6.5 SATA

Internal hard disks require a fast interface to a PC. In 1986, Western Digital introduced the *Integrated Drive Electronics* (IDE) interface, which evolved into the *AT Attachment* (ATA) standard. The standard uses a bulky 40 or 80-wire ribbon cable with a maximum length of 18″ to send data at 16–133 MB/s.

Table e9.11 802.11 Wi-Fi protocols

Protocol	Release	Frequency Band (GHz)	Data Rate (Mb/s)	Range (m)
802.11b	1999	2.4	5.5–11	35
802.11g	2003	2.4	6–54	38
802.11n	2009	2.4/5	7.2–150	70
802.11ac	2013	5	433+	variable

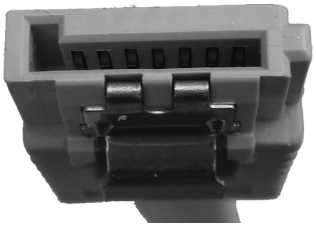


Figure e9.50 SATA cable

ATA has been supplanted by Serial ATA (SATA), which uses high-speed serial links to run at 1.5, 3, or 6 Gb/s over a more convenient 7-conductor cable shown in Figure e9.50. The fastest solid-state drives in 2015 exceed 500 MB/s of bandwidth, taking full advantage of SATA.

A related standard is Serial Attached SCSI (SAS), an evolution of the parallel SCSI (Small Computer System Interface). SAS offers performance comparable to SATA and supports longer cables; it is common in server computers.

9.6.6 Interfacing to a PC

All of the PC I/O standards described so far are optimized for high performance and ease of attachment but are difficult to implement in hardware. Engineers and scientists often need a way to connect a PC to external circuitry, such as sensors, actuators, microcontrollers, or FPGAs. The serial connection described in Section 9.3.4.2 is sufficient for a low-speed connection to a microcontroller with a UART. This section describes two more means: data acquisition systems, and USB links.

9.6.6.1 Data Acquisition Systems

Data Acquisition Systems (DAQs) connect a computer to the real world using multiple channels of analog and/or digital I/O. DAQs are now commonly available as USB devices, making them easy to install. National Instruments (NI) is a leading DAQ manufacturer.

High-performance DAQ prices tend to run into the thousands of dollars, mostly because the market is small and has limited competition. Fortunately, NI sells their handy myDAQ system at a student discount price of $200 including their LabVIEW software. Figure e9.51 shows a myDAQ. It has two analog channels capable of input and output

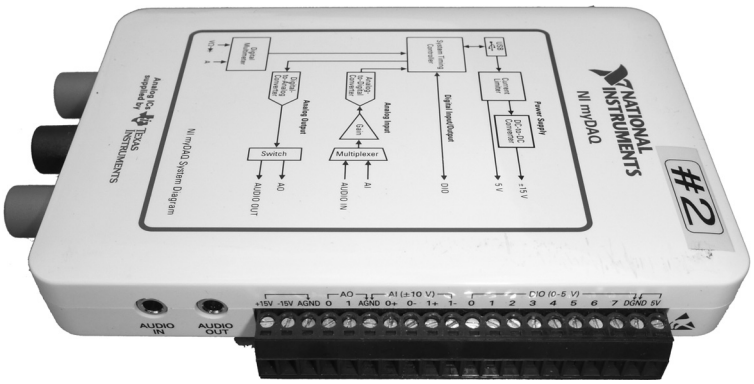


Figure e9.51 NI myDAQ

at 200 ksamples/sec with a 16-bit resolution and ±10 V dynamic range. These channels can be configured to operate as an oscilloscope and signal generator. It also has eight digital input and output lines compatible with 3.3 and 5 V systems. Moreover, it generates 5, 15, and −15 V power supply outputs and includes a digital multimeter capable of measuring voltage, current, and resistance. Thus, the myDAQ can replace an entire bench of test and measurement equipment while simultaneously offering automated data logging.

Most NI DAQs are controlled with LabVIEW, NI's graphical language for designing measurement and control systems. Some DAQs can also be controlled from C programs using the LabWindows environment, from Microsoft .NET applications using the Measurement Studio environment, or from Matlab using the Data Acquisition Toolbox.

9.6.6.2 USB Links

An increasing variety of products now provide simple, inexpensive digital links between PCs and external hardware over USB. These products contain predeveloped drivers and libraries, allowing the user to easily write a program on the PC that blasts data to and from an FPGA or microcontroller.

FTDI is a leading vendor for such systems. For example, the FTDI C232HM-DDHSL USB to Multi-Protocol Synchronous Serial Engine (MPSSE) cable shown in Figure e9.52 provides a USB jack at one end and, at the other end, an SPI interface operating at up to 30 Mb/s, along with 3.3 V power and four general purpose I/O pins. Figure e9.53 shows an example of connecting a PC to an FPGA using the cable. The cable can optionally supply 3.3 V power to the FPGA. The three SPI pins connect to an FPGA slave device like the one from Example e9.4. The figure also shows one of the GPIO pins used to drive an LED.

The PC requires the D2XX dynamically linked library driver to be installed. You can then write a C program using the library to send data over the cable.

Figure e9.52 FTDI USB to MPSSE cable
(© 2012 by FTDI; reprinted with permission.)

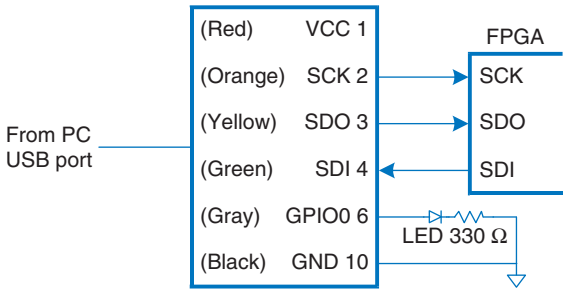


Figure e9.53 C232HM-DDHSL USB to MPSESE interface from PC to FPGA

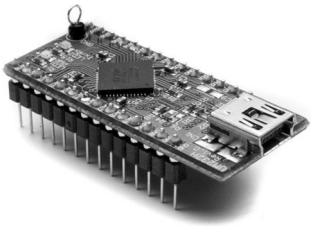


Figure e9.54 FTDI UM232H module
(© 2012 by FTDI; reprinted with permission.)

If an even faster connection is required, the FTDI UM232H module shown in Figure e9.54 links a PC's USB port to an 8-bit synchronous parallel interface operating up to 40 MB/s.

9.7 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. PCs and advanced microcontrollers support more complex I/O standards including USB, Ethernet, and SATA.

This chapter has provided many specific examples of I/O using the Raspberry Pi. Embedded system designers continually encounter new processors and peripherals. The general principal for 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 and then 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.