8

APPLICATIONS OF THE PROPELLER EXPERIMENT CONTROLLER

This chapter describes some applications of the Propeller Experiment Controller (PEC). First, it describes how to connect the Propeller to some common types of equipment and how to interface with this equipment using the PEC's Experimental Event and Experimental Functions objects. Second, it demonstrates several different variations of the PEC's standard workflow, as well as how the workflow can be adapted to many purposes. Finally, this chapter provides an overview of several of our implementations of the PEC in order to provide a greater understanding of the capabilities of the PEC. Variations and expansions of these applications will provide the user with a number of possibilities. It is up to the user to decide what implementation of the PEC will work best for his or her purpose.

Connecting Devices

The Propeller can interface with a wide variety of devices through its I/O pins. This section provides an overview of common devices that can interface with the Propeller, and how to use them with the PEC software. Although we will describe how to connect the devices to the Propeller, this section does not provide detailed information on building circuits. Many electronics texts (Ashby, 2011; Avery et al., 2010; Scherz and Monk, 2016) and hobby electronics websites (see Appendix B) provide detailed information in this area. Generally, we recommend using pin headers to connect the Propeller to devices to jumper cables or matching pin headers. Most of the development boards described in Chapter 4 make this an easy option. Permanent or temporary circuits can be built on solderless breadboards, however, soldering a circuit into a **protoboard** will provide a more secure connection. The materials mentioned in this section can also be purchased from hobby and industrial electronics vendors (see Appendix B).

Digital Input Devices

Many types of digital input devices can be used with the Propeller. The simplest form of digital input device is the **mechanical switch**. This includes push-buttons, toggle switches, rotary switches, and microswitches. Microswitches are especially common in behavioral research equipment. Levers for small animals such as pigeons and rats can be made from lever-actuated microswitches with little to no modification. Other devices, such as weight sensitive plates, can be easily made from a combination of several microswitches. Slightly more complex devices, such as omnidirectional levers, often are designed to activate internal microswitches and can be purchased from industrial electronics suppliers. Mechanical switches are also excellent for building interfaces to allow experimental parameters to be controlled by the user.

Most mechanical switches are very easy to use, requiring only a few connections. To use a switch with the Propeller, connect the common terminal, often labeled as C or COM, to the Propeller's 3.3-volt pin. You may need to solder a wire to the terminal for a secure connection. Note that you can indirectly connect the common terminal to the 3.3-volt pin through some other connection to the 3.3-volt pin, such as the power rails on a breadboard. Next, connect either the normally open (NO) terminal, or the normally closed (NC) terminal, to the Propeller's ground pin through a 10 kiloohm resistor. As before, you can indirectly connect the terminal to ground. Make another connection from the NO or NC terminal to the desired I/O pin. This connection can be made anywhere before the resistor. Use the NO terminal if you want the Propeller to detect an input when the switch is pressed. When the switch is activated, current will flow from the 3.3-volt connection to the Propeller's I/O pin. If you use the NC terminal, current flows when the switch is not activated. Most switches are connected in this manner, but the terminals may not always be labeled, and some switches may not provide the option to use both NO and NC terminals. Push-buttons differ slightly. They have two pairs of terminals. You will only need to use one of the terminal pairs; the other is provided so additional circuits can be activated by the button, and for structural support on a breadboard or protoboard. If a switch is not clearly labeled, the easiest way to determine how the terminals function is to use the continuity feature of a multimeter. For any type of switch, Experimental Event's Detect method can be used to determine the state of the switch.

Digital infrared (IR) sensors are useful in applications where you want to record a subject's activity without the subject being aware, such as when the subject walks into an area. These sensors are also useful when you want to record a behavior but remove the force requirement of physically activating a switch. For example, the nose-poke response of many rodent apparatuses does not require force; instead the nose-poke is detected by an infrared sensor. Infrared light can also pass through glass and water, making infrared sensors useful for detecting the behavior of fish and other aquatic species in aquariums, while protecting the electronics from water. Digital IR sensors come in two basic forms: infrared beam-break sensors and reflective infrared sensors.

Infrared beam-break sensors are created with an IR receiver and IR emitter pair. When aligned properly, this creates a beam of light that is invisible to most species. Any activity of the subject that breaks the beam can be recorded. Unfortunately, interference from ambient IR light, such as sunlight, is a problem for many receivers. Modulated IR beams provide a solution. A modulated IR receiver only detects IR light turning on and off at a particular frequency, usually 38,000 times a second (38 kHz modulation). The receiver is therefore less sensitive to ambient IR light that does not modulate, such as sunlight. You will need an IR emitter, often called an IR LED, and a special modulated IR receiver to create a modulated IR beam-break sensor. To create the sensor, first connect the anode of the IR emitter to an I/O pin on the PEC through a current limiting resistor. Larger resistor values will create a dimmer, smaller infrared beam that can be broken by small objects, while smaller resistor values will create a brighter infrared beam that can span longer distances. Use an online LED resistor calculator to determine the smallest possible resistor value (see Appendix B). Connect the cathode of the infrared LED to ground. Experimental Functions's Frequency Generator methods can be used to generate a 38 kHz frequency on the emitter. Modulated IR receivers typically have three pins. Connect the ground pin to the PEC's ground, the power pin to 5 VDC, and the output pin to an I/O pin on the Propeller through a 220-ohm resistor. This is an instance where it is useful to have a development board with both 3.3 VDC and 5 VDC regulators. When modulated IR light from the emitter hits the receiver, current will flow from the collector to the emitter, activating the Propeller's I/O pin in the process. Keep in mind that placement of the emitter and receiver is important, as multiple beams in close proximity may interfere with proper detection of beam-breaks. Positioning the components correctly can be made easier using digital cameras, as many, including those found in cellular phones, can detect IR light. Once the IR sensor is installed, Experimental Event's detect method can be used to detect when the beam is broken. We find that using the SetDebounce method to increase the default debounce time to 100 ms is often useful with IR sensors that produce false detections when an object is very close to the beam.

Reflective infrared sensors shine light forward and then detect light that reflects from a surface, usually in close proximity to the sensor. These sensors are excellent at detecting when a subject approaches a certain area, or for building response devices that do not require force, such as a nose-poke hole for a rodent. Pololu offers many easy to connect boards with Sharp brand reflective modulated digital IR sensors that operate in ranges from 0.5 to 15 cm. To connect the Pololu Sharp reflective IR sensors to the Propeller, simply connect the power pin of the sensor to 3.3 VDC, the ground pin to ground, and the output pin to an I/O pin. The output from the sensor is inverted, meaning that the I/O pin will receive current when the sensor does not detect an object, and will receive current when an object is detected. Because the output is inverted, you should use the DetectInverted method instead of the Detect method with this sensor. As with IR beam-break sensors, increasing the debounce time is sometimes useful at reducing false detections.

Analog Input Devices

The Propeller's I/O pins are digital, and do not directly support analog input. There are, however, several ways to obtain analog input with the Propeller. One of the most convenient manners to obtain analog input with the Propeller is to use an external ADC, such as the MCP3208. The MCP3208 provides eight analog input channels with 12-bit resolution, meaning that the value reported from each channel can range from 0 to 4,095. To use the MCP3208 with the PEC, connect the DIN and DOUT pins on the MCP3208 to one I/O pin on the PEC. Connect the CLK pin on the MCP3208 to a second pin on the PEC. Connect the CS pin on the MCP3208 to a third pin on the PEC. The MCP3208's Vdd and Vref pins can be connected to a 3.3 or 5 VDC power supply; this connection determines the maximum voltage the analog input channels can read. Then, the MCP3208's AGND and DGND pins can be connected to the PEC's ground pin. After the connections are made, Parallax's MCP3208 object (Gracey, 2005) can be used to read analog inputs. This object requires one cog. Data from the analog inputs can be recorded and saved using the raw data event, and the RecordRawData and SaveRawData methods. Once a method to read analog inputs is obtained, a variety of new input devices become available. For example, analog inputs can be used to detect sound through an electret microphone, photoresistors can be used to detect light levels, analog infrared proximity sensors can detect how far away an object is, and EMG (electromyography) sensors can be used to detect muscle movement.

Complex Input devices

A variety of complex input devices are also available. These are often specialized sensors that provide information to the Propeller through an I^2C, SPI, or other communication protocol. For example, the SHT15 (Sensirion; Staefa, Switzerland) detects both the humidity and temperature, then communicates this information to the Propeller, or other microcontroller, through two I/O pins. As each complex input device has unique requirements, detailed instructions cannot be provided here. Instead we recommend browsing hobby electronics websites, and look for pre-written programs to control the sensors in Parallax's Propeller Object Exchange. Then, manual events or raw data events may be useful to save information regarding the input.

High Voltage Inputs

It is important to note that the Propeller is a 3.3 VDC system. Any input device that produces more than 3.3 VDC will damage the Propeller. Alternating current (AC) may also damage the Propeller. Many of the useful input devices found at commercial and hobby electronics vendors produce 5 VDC signals. In addition to reading the documentation for a device, it is also often useful to check the voltage level with a multimeter. Voltage can be reduced to a safe level by using a pair of resistors to create a voltage divider. Online voltage divider calculators can provide resistor values needed to reduce the voltage level to 3.3 VDC (see Appendix B). Special chips, such as logic level-converters or optocouplers, can be used to convert between high and low voltages. Transistor circuits (see the output section) can also be used to decrease a high voltage input to 3.3 VDC safe for the Propeller's I/O pins.

Digital Output Devices

The Propeller can also interface with a variety of output devices, the simplest of which are digital outputs that can be turned on and off directly by an I/O pin. As the Propeller's I/O pins generate 3.3 VDC, it is best to use output devices that operate at this voltage. Many 5 VDC output devices will operate at 3.3 VDC. Check the datasheet to be certain of the device requirements.

One common type of output device is the LED. These small lights can be used as stimuli for many conditioning procedures and can also be used for general lighting. In most cases, LEDs can be powered directly by the Propeller's I/O pins. Simply connect the longer leg of the LED, the anode, to an I/O pin through a small resistor. A 220-ohm resistor will work for most LEDs. Online LED resistor calculators can be used to find more specific resistance values (see Appendix B). Connect the shorter leg of the LED, the cathode, to the Propeller's ground pin. LEDs can then be controlled with the TurnOn and TurnOff methods. Multicolor LEDs are also available that contain two or three colors inside one LED case. The LEDs often share a common anode or cathode, with the other terminals being independent. Once the terminals are identified, the individual LEDs inside the multicolor LED can be connected and controlled just like single LEDs. Turning on multiple LEDs in multicolor LED can result in many color combinations. For example, using the TurnOn method to activate both the red and blue LEDs in a RGB (red, green, blue) LED will cause the total output of the RGB LED to appear purple.

The Propeller can also generate audio signals through digital output devices. **Piezo speakers** offer one convenient way to incorporate audio signals. As with LEDs, connect the longer leg of the speaker to an I/O pin and connect the shorter leg to ground. Occasionally, the legs are the same length; a plus symbol will denote the end to connect to the I/O pin. Most piezo speakers will not produce a tone simply by powering them. Instead, use the Frequency Generator methods to produce tones. Manual events can then be used in conjunction with frequency generation to record when the frequencies start and stop. Many breakout boards also allow the Propeller to use to standard audio connections. For louder audio signals, a separate amplifier board may be needed. In addition to generating basic square wave frequencies, several objects are available on Parallax's Propeller Object Exchange to generate other wave forms, synthesize speech, and play wav audio files. Playing wav files, however, often requires files to be hosted on a separate SD card. As the PEC uses the SD card for data output, an external audio player board may be preferred.

High Voltage Digital Outputs

Some digital output devices require more power than can be provided by the Propeller's 3.3 VDC I/O pins. For example, many pellet dispensers are powered by motors that require 12 VDC power supplies. Vibrating motors also make excellent stimuli and can be used as noisemakers when attached to hollow objects. Cooling fans can act as reinforcers for hot animals. Many lights used for stimuli or to illuminate an experimental apparatus also require more power than the Propeller can directly provide. Fortunately, there are several ways the Propeller can indirectly provide ample power for these types of devices.

Transistors are one option to allow the Propeller to indirectly provide more DC voltage to a device. The general function of a transistor is to provide a higher voltage to one device when it receives a lower voltage signal from a second device, in this case the Propeller. The transistor used will vary greatly depending on the requirements of the higher voltage device, but we recommend the common 2N2222A or 2N3904 transistors (multiple manufactures) for most purposes. Review the transistor's datasheet to determine if it fits your application. To use a transistor with the Propeller, connect the transistor pin labeled base to an I/O pin on the Propeller through a 10-kiloohm resistor. Then, connect the emitter pin of the transistor to the ground of the higher voltage power supply, and the collector of the transistor to the positive end of the higher voltage power supply.

Insert your device between the positive end of the higher voltage power supply and the transistor's collector. When the I/O pin turns on and powers the base, the transistor connects the collector and emitter to power the device. Convenient integrated circuits, such as the ULN2803A Darlington array (multiple manufactures), contain a series of transistors in one small package. The ULN2803A also contains internal resistors before the base of each transistor, so an additional resistor is not typically required. It is, however, always useful to check the datasheet to be sure the internal resistor is sufficient. Devices that produce electromagnetic fields, such as DC motors and DC fans require a slight addition to the circuit in order to protect the Propeller. In this case, connect a diode in parallel with the device, with the cathode connected to the power source and the anode connected to the collector. The ULN2803A already includes this protective diode, but again, check the datasheet to be safe. To avoid some of complexities of creating a transistor circuit, hobby electronics vendors also offer transistor breakout boards for quick connections.

Electromechanical relays can also be used to control output devices that require more power than the Propeller can provide. Some relays can also control AC devices. Unfortunately, relays can be rather large, and many produce an audible click when changing states, and so may not be preferable to transistors in all cases. Relays are connected to the Propeller using the same transistor and diode circuit used for motors and fans. Many relay breakout boards are available that already have this protection and control circuitry built in.

Transistors and relays are particularly useful at controlling external devices by controlling their power source. For example, a Super-Feeder (Super-Feed Enterprise; Burleson, Texas; Super-Feeder.com) can be modified for experimental use by soldering a wire between the two terminals of the manual feed button on the front of the feeder. The feeder now functions as if the manual feed button is always pressed and dispenses food any time it receives power. The Propeller can then control the feeder by using a relay to turn on and off power to the feeder using the TurnOn and TurnOff methods. A relay or transistor circuit can simply be spliced into the feeder's power supply cable to give the Propeller control of the feeder.

Analog Output Devices

Although the Propeller's I/O pins are digital, the Pulse-Width Modulation (PWM) methods are excellent emulations of analog voltage. For most purposes, PWM is an adequate replacement for true analog output. Using PWM with LEDs allows the lights to be dimmed. With multicolor LEDs, combinations of bright and dim individual LEDs can result in a much greater variety of colors. For audio generation, at low frequencies the PWM methods can be used to produce tones, and the duty cycle can be used to adjust timbre. Analog voltage can also be emulated with PWM for transistor-controlled devices. For example, using PWM with a transistor-controlled motor can reduce motor speed. Relays however, cannot be toggled as fast as transistors; PWM may lead to relay failure. In the unusual case that a true analog output is required, specialized resistor/capacitor low-pass filter circuits can be designed to filter the PWM signal and create analog output. However, this is a more complex topic than finding the best value of a resistor for an LED or building a voltage divider out of LEDs and should only be implemented by those with more electronics experience. A digital-to-analog converter chip, such as the MCP4725 (Microchip; Chandler, Arizona), can also be used to produce true analog output. For any analog output device, the manual event and raw data event types can be used to record information about the state of the outputs. For example, two manual events may be used to record when a motor's speed is low (one event) or high (a second event). Alternatively, the raw data event may be used to record the specific speed (duty cycle) of a PWM motor.

Complex Output Devices

A large variety of complex output devices are available for the PEC. Like complex inputs, the requirements vary greatly. Thus, the best approach to finding complex output devices is to browse hobby electronics vendors and Parallax's Propeller Object Exchange for appropriate driver software. A few complex devices, however, are worth discussing in more detail.

The **WS2812 LED** (World Semi; Daling Village, China) is a multicolor LED with a red, blue, and green LED in a small casing. These are often referred to as NeoPixels at hobby electronics vendors. Instead of controlling each LED with a separate I/O pin, a single I/O pin can send a signal to the WS2812 using a special communication protocol to determine the brightness value of each individual LED. This enables the Propeller to specify the brightness of each of the three LED colors inside the WS2812, allowing for any color to be created. The WS2182 can then pass information on to another WS2812, and that WS2812 to another, leading to a long chain of WS2182s that can be controlled by a single I/O pin on the Propeller. As the WS2812s can operate in a long series, they are often sold packed together in rings, chains, or large arrays. While some microcontrollers struggle with the communication protocol required by the WS2812, the Propeller can specialize a cog dedicated to controlling an array of WS2812s, while the rest of the cogs can continue operation as normal. To connect the WS2812, connect the power and ground pins to a 5 VDC, 1 amp power supply. This should be separate from the Propeller's power supply. Next, connect a 1000 μF capacitor rated for at least 6.3 volts across the power and ground connections. Finally, connect the data input pin of the WS2182 to an I/O pin on the Propeller through a 300- to 500-ohm resistor.

Several objects are available on Parallax's Propeller Object Exchange to control the WS2812s; we prefer the JM_WS2812 object (McPhalen, 2016). The JM_WS2812 uses a start method that requires the I/O pin connected to the WS2812, and the number of WS2812s in the series as parameters. This method launches a constantly updating WS2812 driver in a new cog. Then, the brightness and color values of each WS2812 in the series can be set at any time by the main cog. When using RGB LEDs like the WS2812, the PEC's manual event methods can be used to note when an individual color (e.g., blue) or combination color (e.g., yellow) starts and stops being presented. The raw data events can be used to record the exact value of each LED for more precision.

In addition to controlling standard motors through transistors, the PEC can also control servo motors. Servo motors typically have a limited range of rotation but can be instructed to travel to specific positions within that range. Because of their ability to be precisely controlled, servos can be used for a variety of applications, including bringing reinforcement or stimuli into range of a subject in an experimental apparatus, or acting as the joints in robotic limbs. Servos can be connected to the Propeller by connecting their power pin, usually a red wire, and the ground pin, usually a black wire, to an external power supply. Then the input pin, usually white wire, can be connected an I/O pin on the Propeller. Although smaller servos can be powered by 3.3 VDC or 5 VDC, they can draw large amounts of power rapidly, and it is best to provide servos with a separate power supply from the Propeller. Experimental Functions's StartPWM method, along with the Servo method, can potentially be used to control up to 32 servos simultaneously. Some special considerations will be needed when connecting servos to pins 28 to 31 as the Propeller uses these pins during boot-up.

Stepper motors offer another method to control motion. Instead of spinning freely like a standard motor or moving to fixed position in a limited range like a servo, stepper motors rotate in precise steps. This allows the rotation of a stepper motor to be controlled very precisely. Programs that count the number of steps are also able to calculate how much a stepper motor has rotated. Stepper motors are typically used in applications where full rotation and precise control are needed. Because of their unique design, stepper motors also have unique and rather complicated control requirements; and thus, the PEC does not support them directly as with standard motors and servo motors. To control stepper motors, it is often best to rely on an external stepper motor driver board, such as Pololu's DRV8825 stepper motor driver. After an appropriate driver board is found, software to interface with the driver board will also be required. Check Parallax's Propeller Object Exchange for stepper motor control objects. Once control hardware and software are obtained, stepper motors can be used for many projects requiring precision rotation. For example, we designed a stepper motor driven syringe pump that can deliver precise amounts of liquid or paste to animals. Full instructions, part lists, and programs are available, at CAVarnon.com/SyringePump. The ability of the stepper motor to be very precise allows for the pump to reliably deliver very small quantities of fluid. For this pump, a 15-step rotation results in about 0.25 mL of water delivered. This type of precision is characteristic of stepper motor actuated devices. However, it comes at a cost of increased complexity compared to a standard motor or servo motor. As with other types of complex output, manual events and raw data events can be used to record information about the output.

The Propeller can also connect to many types of display devices. One convenient device is Parallax's Serial LCD screen (Parallax Inc.; Rocklin, California), available in a 2 by 16 character, or 4 by 16 character, backlit or unlit display. The LCD screen can be connected to the Propeller by connecting the Vdd pin to 5 VDC, the GND pin to ground, and the RX pin to an I/O pin on the Propeller. The LCD screen can then be controlled by Parallax's Serial LCD object (Williams and Martin, 2006). Like the Parallax Serial Terminal, the LCD screen is started with an Init method. Then, characters can be displayed with the Putc method, and strings can be displayed with the Str method. Other methods are also available, but the object does not directly support displaying integers. Experimental Functions's ToStr method can be used to convert integers to strings for display on the LCD screen. The LCD screen, along the Parallax's Serial Terminal, are both excellent at providing information about an experiment, as well as testing and debugging programs. For example, if an event onset is detected, the LCD screen can display the time of the event using a combination of the LCD's Str method, and Experimental Functions's ToStr method. Parallax's Propeller Object Exchange also provides objects to generate composite or VGA signals for television screens or computer monitors. Compared to a personal computer, the Propeller's video capabilities are limited, and are best used for text and simple graphics. Displaying basic graphics on a video screen, however, would be adequate for presenting video stimuli in an experiment.

Workflows

While the PEC was designed around a specific program workflow, clever use of Experimental Event, Experimental Functions, and basic Spin instructions can be utilized to create a variety of alternative workflows suitable for a wide range of experiments and automation projects. It is up to the user to decide what type of program workflow is best for their project. The examples below show some alternatives to the standard workflow that may be useful.

Continuous Background Recording of Events

The Propeller's multicore, eight-cog architecture can be used to record background information while an experiment is conducted. As an example, first consider how the program in Figure 7.25 might be modified to collect information about heart rate in the background. Recall from the previous chapter that this program delivers food on each lever onset, and also records heart rate when the lever is pressed. Recording one event contingent on another, in this case recording heart rate contingent on a lever-press, is useful in many circumstances. However, it may also be useful to continuously record information about some event in the background, regardless of the state of other events.

Figure 8.1 shows a modification of the program shown in Figure 7.25. In this example, heart rate is no longer recorded at each lever onset, and is instead recorded every 2 seconds using an additional cog. The COGNEW instruction is used to launch the method, HeartRateCog, before the main repeat loop begins. The new cog will use the stack space reserved by HeartRateCogStack, this should be an array of around 100 longs. The HeartRateCog uses an infinite repeat loop with a synchronized pause. Inside the loop, the variable BPM is assigned to be the beats per minute value obtained from the HRS.BMP method. In order for the BMP variable to hold heart rate data, it should be byte- or word-sized, depending on the heart rate of the species being studied. Next, the variable NewData is set to 1. This variable will act as a flag to tell the main cog that data have been collected. As NewData will only be 0 or 1, it should be a byte-sized variable. The cog then executes a synchronized pause of 2 seconds. In the Main method, the repeat loop detects lever-presses, delivers food, and records events to the memory file just as in the previous version of the program. Now, however, if the NewData variable is 1, the heart rate will be recorded, and NewData will be set back to 0. This results in heart rate only being recorded when a new measurement is obtained. Note that the RecordRawData method is not placed in the HeartRateCog to ensure that multiple cogs do not try to write to the SD card at the same time.

Using a second cog to record background information is efficient and relatively easy to execute. Although a clever programmer could find a way to perform the same functions with a single cog, there is simply no reason to do so with so many cogs to spare. Additionally, as background tasks become more complicated, it becomes more difficult to implement everything in a single cog, and the performance of the main program loop may suffer. The multiple-cog approach is almost always a better solution, except in rare cases where all cogs are occupied.

Controlling Multiple Apparatuses

The Propeller can control many more events than have been shown in the previous examples, making it easy for a single Propeller to control multiple independent apparatuses. Controlling multiple apparatuses only requires adding more events to a program, along with more conditionals to control those events. Figure 8.2 shows an example of a multi-apparatus approach. The basic program builds upon the previous examples where food is delivered when a lever is pressed. The main repeat loop now contains two detect methods, one for each lever event. This

```
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    Pulse.DeclareRawData(EXP.ClockID)
    COGNEW(HeartRateCog, @HeartRateCogStack)
    Start := EXP.Time(Start)
    REPEAT UNTIL EXP.Time(Start) > SessionDuration
        EXP.Record(Lever.Detect, Lever.ID, EXP.Time(Start))
        IF Lever.State == Onset
            EXP.Record(Feeder.TurnOn, Feeder.ID, EXP.Time(Start))
            FeederStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            EXP.Record(Feeder.TurnOff, Feeder.ID, EXP.Time(Start))
        IF NewData == 1
            EXP.RecordRawData(BPM, Pulse.ID, EXP.Time(Start))
            NewData := 0
    EXP.Record(Feeder.TurnOff, Feeder.ID, EXP.Time(Start))
    IF Lever.State == Onset OR Lever.State == On
        EXP.Record(Offset, Lever.ID, EXP.Time(Start))
    EXP.PrepareDataOutput
    EXP.SaveData(Lever.ID, STRING("Lever"))
    EXP.SaveData(Feeder.ID, STRING("Feeder"))
    EXP.PrepareDataOutputForRawData
    EXP.SaveRawData(Pulse.ID, STRING("Heart rate (bpm)"))
    EXP.Shutdown
PUB HeartRateCog
    EXP.StartSync
    REPEAT
        BPM := HRS.BPM
        NewData := 1
        Exp.SyncPause(2000)
```

Figure 8.1: Continuous background recording of events example program.

is followed by two conditionals to evaluate each lever's state, then two conditionals to deactivate the feeders after a specified duration. There is no reason why the program could not be further modified to support additional apparatuses. It is also important to note that each subject does not have to experience the same experimental parameters. It is up to the user to decide how to implement a multi-apparatus workflow. Alternatively, the same basic concept shown here could be used to provide a single subject with multiple contingencies. To reduce figure size, the CON, VAR, and OBJ blocks are omitted.

```
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    Lever1.DeclareInput(Lever1Pin, EXP.ClockID)
    Lever2.DeclareInput(Lever2Pin, EXP.ClockID)
    Feeder1.DeclareOutput(Feeder1Pin, EXP.ClockID)
    Feeder2.DeclareOutput(Feeder2Pin, EXP.ClockID)
    Start := EXP.Time(Start)
    REPEAT UNTIL EXP.Time(Start) > SessionDuration
        EXP.Record(Lever1.Detect, Lever1.ID, EXP.Time(Start))
        EXP.Record(Lever2.Detect, Lever2.ID, EXP.Time(Start))
        IF Lever1.State == Onset
            EXP.Record(Feeder1.TurnOn, Feeder1.ID, EXP.Time(Start))
            Feeder1Start := EXP.Time(0)
        IF Lever2.State == Onset
            EXP.Record(Feeder2.TurnOn, Feeder2.ID, EXP.Time(Start))
            Feeder2Start := EXP.Time(0)
        IF EXP.Time(Feeder1Start) > FeederDuration
            EXP.Record(Feeder1.TurnOff, Feeder1.ID, EXP.Time(Start))
        IF EXP.Time(Feeder2Start) > Feeder2Duration
            EXP.Record(Feeder2.TurnOff, Feeder2.ID, EXP.Time(Start))
    EXP.Record(Feeder1.TurnOff, Feeder.1ID, EXP.Time(Start))
    EXP.Record(Feeder2.TurnOff, Feeder2.ID, EXP.Time(Start))
    IF Lever1.State == Onset OR Lever1.State == On
        EXP.Record(Offset, Lever1.ID, EXP.Time(Start))
    IF Lever2.State == Onset OR Lever2.State == On
        EXP.Record(Offset, Lever2.ID, EXP.Time(Start))
```

Figure 8.2: Multi-apparatus example program.

There are a few different ways that data from multiple apparatuses can be saved. Figure 8.2 did not include any SaveData methods. Instead, two options are presented below. Figure 8.3 shows the standard form of saving each event to the same data spreadsheet. Figure 8.4, however, saves the data from each apparatus to a separate data spreadsheet. The PrepareCustomDataOutput method is used to prepare a file for apparatus 1, called "Box 1.csv" in the program to meet the file name length requirements. Then, the data for apparatus 1 are saved. Next, a new data spreadsheet is created for apparatus 2, and its data are saved. The basic method outlined here could also be used to save events from an experiment with a single subject in the same data file. For example, a subject might be exposed to two different conditions, and data from each condition might be saved separately.

```
EXP.PrepareDataOutput
EXP.SaveData(Lever1.ID, STRING("Lever 1"))
EXP.SaveData(Lever2.ID, STRING("Lever 2"))
EXP.SaveData(Feeder1.ID, STRING("Feeder 1"))
EXP.SaveData(Feeder2.ID, STRING("Feeder 2"))
EXP.Shutdown
```

Figure 8.3: Saving data from multiple apparatuses in one file example program.

```
EXP.PrepareCustomDataOutput(STRING("Box 1.csv"))
EXP.SaveData(Lever1.ID, STRING("Lever"))
EXP.SaveData(Feeder1.ID, STRING("Feeder"))
EXP.PrepareCustomDataOutput(STRING("Box 2.csv"))
EXP.SaveData(Lever2.ID, STRING("Lever"))
EXP.SaveData(Feeder2.ID, STRING("Feeder"))
EXP.Shutdown
```

Figure 8.4: Saving data from multiple apparatuses in separate files example program.

Saving Data from Multiple Sessions

Most applications of the PEC described so far run a single session, save data, then turn off. This requires the user to collect the data spreadsheet from the SD card before resetting the Propeller to run another session. For many experiments, this is not an issue. However, it may be useful in some cases to run several consecutive sessions without needing to transfer data from the SD card to the computer. The PEC can easily implement this type of workflow, although this requires a more advanced understanding of reading and writing files on the SD card.

Figure 8.5 shows such an example. To reduce figure size, the CON, VAR, and OBJ blocks are omitted. This program is based on the previous examples where food is delivered on each lever-press but also is able to read a file on the SD card, named "settings.txt", to determine which session is being run. It adjusts the data spreadsheet name accordingly. The program also updates the session number saved on the settings file after the experiment is complete. This is all accomplished by clever use of the SD card methods. The Main method first mounts the SD card. Then, it opens the settings file. The variable, DataFile, has been created in the DAT block to hold the name of the data spreadsheet. Instead of the default "data.csv", the string at DataFile is "dataN.csv". The "N" will be replaced with the current session number. Once the file is opened, the SDReadCharacter method reads the first character in the file, and assigns it to DataFile[4]. This replaces the "N" in "dataN.csv" with a number that was found in the settings file. Next, the program needs to get ready to run the experiment. The StartExperiment_NoData method is used to start the system clock and generate a random number seed.

The SD card has already been mounted, so the regular StartExperiment method that mounts the SD card is not needed. The EXP.CreateMemoryFile method is used to create the default memory file. This is normally done by the regular StartExperiment method, so it needs to be manually created here. The program can then run as normal until the session ends.

```
PUB Main
    EXP.SDMountCard(DO, CLK, DI, CS)
    EXP.SDFileOpen(STRING("settings.txt"), "r")
    DataFile[4] := EXP.SDReadCharacter
    EXP.StartExperiment_NoData
    EXP.CreateMemoryFile(@MemoryFile)
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    Start := EXP.Time(Start)
    REPEAT UNTIL EXP.Time(Start) > SessionDuration
        EXP.Record(Lever.Detect, Lever.ID, EXP.Time(Start))
        IF Lever.State == Onset
            EXP.Record(Feeder.TurnOn, Feeder.ID, EXP.Time(Start))
            FeederStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            EXP.Record(Feeder.TurnOff, Feeder.ID, EXP.Time(Start))
    EXP.Record(Feeder.TurnOff, Feeder.ID, EXP.Time(Start))
    IF Lever.State == Onset OR Lever.State == On
        EXP.Record(Offset, Lever.ID, EXP.Time(Start))
    EXP.PrepareCustomDataOutput(@DataFile)
    EXP.SaveCustomData(Lever.ID, STRING("Lever"), @MemoryFile, @DataFile)
    EXP.SaveCustomData(Feeder.ID, STRING("Feeder"), @MemoryFile, @DataFile)
    EXP.SDFileOpen(STRING("settings.txt"), "w")
    EXP.SDWriteCharacter(DataFile[4]+1)
    EXP.Shutdown
DAT
    MemoryFile  BYTE    "memory.txt", 0
    DataFile    BYTE    "dataN.csv", 0
```

Figure 8.5: Multi-session example program.

After the session ends, the data need to be saved. The PrepareCustomDataOutput method creates a data spreadsheet named "dataN.csv", where "N" has already been replaced with the session number. The SaveCustomData method is then used to save data to the custom named data spreadsheet. Next, the settings file is opened again, this time for writing. Opening the file in writing mode causes the previous contents to be overwritten. The SDWriteCharacter method writes a single character in this file, DataFile[4]+1. Although adding is normally reserved for integers, not characters, the character numbers are represented by bytes, so addition is still possible. If the character first read from the settings file was "0", DataFile will be "data0.csv". Adding 1 to DataFile[4] will actually change this to "data1.csv". This is because the *character* "0" is one less than the *character* "1" in terms of the *integer* value of the byte. In this case, "0" is 48 in decimal, and "1" is 49 in decimal. Therefore, "0" + 1 is actually 48 + 1, which equals 49, or "1". (See the Number and String Conversion Methods in Chapter 7 as well as the character charts in Appendix C for more details on converting numbers to characters and strings.) Finally, the Shutdown method unmounts the SD card, which saves the settings file. The Shutdown method also stops the system clock. The result of this program repeatedly reading, incrementing, and writing the first character of "settings.txt" means that the Propeller will save a series of uniquely named data spreadsheets each time it is reset after an experiment session. Optimally, the settings file will contain only a "0" on first use. The data spreadsheet names can then increment from 0 to 9. After 9, the settings file should be reset, as the character addition will no longer work (e.g., "9" + 1 = ":"). Although this example is only adequate for incrementing a file from 0 to 9, an extension of these methods could increase the range of data spreadsheets that could be saved. There are also many other possible experimental parameters that could be stored in the settings file.

Noncontinuous Workflows

The PEC was designed to provide continuous recording of events. However, a clever programmer can find ways to implement other types of recording workflows, such as interval recording. Figure 8.6 establishes a standard continuous recording workflow. The program records two events, Area1 and Area2. Imagine these events correspond to a subject entering one side of an apparatus or the other, as detected by some kind of proximity sensor. Perhaps the experimenter wants to monitor the duration spent in these areas across four, 5-minute intervals. Here, the repeat loop continuously records behavior until the time since the experiment started is equal to four, 5-minute interval lengths, or 20 minutes total. If the experimenter wanted to analyze the data in terms of four, 5-minute intervals, she would have to carefully consider how to parse the continuous data, perhaps creating a program on her personal computer to do so.

Figure 8.7 shows a variation of this program that is designed to automatically provide interval data. The repeat loop has been changed greatly. Now, the loop repeats four times, while also incrementing the variable, Interval, from 1 to 4. Inside this repeat loop is another repeat loop. The inner repeat loop continues until the 5-minute interval length expires, using the familiar REPEAT UNIT EXP.TIME(START) > technique. The inner repeat loop simply detects

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    #0, Off,        #1, Onset,      #2, On,         #3, Offset
    #0, DO,         #1, CLK,        #2, DI,         #3, CS
    #5, Area1Pin,                   #7, Area2Pin
    IntervalLength = 300_000
OBJ
    EXP:    "Experimental_Functions"
    Area1:  "Experimental_Event"
    Area2:  "Experimental_Event"
VAR
    LONG    Start
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    Area1.DeclareInput(Area1Pin, EXP.ClockID)
    Area2.DeclareInput(Area2Pin, EXP.ClockID)
    Start := EXP.Time(0)
    REPEAT UNTIL EXP.Time(Start) > IntervalLength * 4
        EXP.Record(Area1.Detect, Area1.ID, EXP.Time(Start))
        EXP.Record(Area2.Detect, Area2.ID, EXP.Time(Start))
    IF Area1.State == Onset OR Area1.State == ON
        EXP.Record(Offset, Area1.ID, EXP.Time(Start))
    IF Area2.State == Onset OR Area2.State == ON
        EXP.Record(Offset, Area2.ID, EXP.Time(Start))
    EXP.PrepareDataOutput
    EXP.SaveData(Area1.ID, STRING("Area1"))
    EXP.SaveData(Area2.ID, STRING("Area2"))
    EXP.ShutDown
```

Figure 8.6: Continuous recording example program.

both events. After the inner repeat loop ends, data from both events are recorded. Here, the RecordRawData method is used to record each event's cumulative duration, as detected by the inner repeat loop. Instead of saving the actual time of an event, the RecordRawData method is used, Interval*1000 is saved. This will cause the RecordRawData method to save the current interval, from 1 to 4, scaled up by 1000. The interval is multiplied by 1000 so when the SaveRawData method interprets Interval as milliseconds, the resulting data saved to the spreadsheet are 1 to 4, instead of 0.001 to 0.004. After the data are recorded, the duration of each area is reset. The program then returns to the start of the outer repeat loop, increments the Interval variable, resets the Start variable, then enters the inner repeat loop again. Data are saved to the data spreadsheet after four, 5-minute intervals have passed using the PrepareDataOutputForRawData and SaveRawData methods. Ultimately, the program saves the duration that each area event occurred at each of the four, 5-minute intervals. Although the data obtained are not as detailed as would be obtained from the example in Figure 8.7, the program is designed to efficiently provide the desired data.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    #0, Off,        #1, Onset,     #2, On,       #3, Offset
    #0, DO,         #1, CLK,       #2, DI,       #3, CS
    #5, Area1Pin,                  #7, Area2Pin
    IntervalLength = 300_000
OBJ
    EXP:   "Experimental_Functions"
    Area1: "Experimental_Event"
    Area2: "Experimental_Event"
VAR
    LONG  Start
    BYTE  Interval
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    Area1.DeclareInput(Area1Pin, EXP.ClockID)
    Area2.DeclareInput(Area2Pin, EXP.ClockID)
    REPEAT Interval FROM 1 TO 4
        Start := EXP.Time(0)
        REPEAT UNTIL EXP.Time(Start) > IntervalLength
            Area1.Detect
            Area2.Detect
        EXP.RecordRawData(Area1.Duration, Area1.ID, Interval*1000)
        EXP.RecordRawData(Area2.Duration, Area2.ID, Interval*1000)
        Area1.SetDuration(0)
        Area2.SetDuration(0)
    EXP.PrepareDataOutputForRawData
    EXP.SaveRawData(Area1.ID, STRING("Area1"))
    EXP.SaveRawData(Area2.ID, STRING("Area2"))
    EXP.ShutDown
```

Figure 8.7: Interval recording example program.

In addition to breaking away from the PEC's standard continuous recording workflow, this program "cheats" in several ways. First, the events Area1 and Area2 are declared as input events. Yet the RecordRawData method is used to record information about them. This works because the RecordRawData method only requires an event ID. Any Declare method provides an event ID. In this case, Area1 and Area2 were declared as input events in order to use the Detect method. The RecordRawData method then uses the IDs already assigned to the events to record data to the memory file. The second way this program "cheats" is by providing Interval*1000 as the EventTime in the RecordRawData method. The RecordRawData method expects a millisecond time, but in truth, all that is required is an integer. This type of "cheating" with the PEC's objects may make many other types of workflows possible. After becoming familiar with the general use of the PEC, opening the Experimental Event and Experimental Functions objects and reading the code comments will provide an even greater understanding, potentially leading to discovering ways to implement novel workflows.

Example Applications of the Propeller Experiment Controller

The PEC has been used in several research and teaching projects. In the following sections, we describe some of the applications of which we have been involved. The example applications here are intended to provide a greater perspective on how the PEC can be used. These examples are not intended to be a comprehensive review of the findings of this research, rather they are illustration of how the PEC might be used in other projects. In the following sections, we describe a sexual conditioning experiment in pigeons, an inexpensive laboratory for teaching principles of conditioning and learning, a fixed-interval schedule of reinforcement experiment in horses, a foraging experiment in wild squirrels, and an aversive conditioning experiment in honey bees.

Sexual Conditioning in Pigeons

The first application of the PEC was to study sexual conditioning in pigeons (*Columba livia*; Varnon, 2013). In this experiment, human-reared and parent-reared pigeons learned to associate a stimulus light with a social stimulus. For the human-reared birds, the social stimulus was access to a human; for the parent-reared birds, the social stimulus was access to their mate. During experiment sessions, the Propeller activated the stimulus light for 30 seconds, then signaled the experimenter, using LEDs outside of the apparatus, to open a door to a stimulus chamber, revealing the social stimulus. The Propeller also recorded the duration the birds spent within 5 and 20 cm of two stimulus chambers, one adjacent to the stimulus light. This experiment used several conditions. A switch was installed on the apparatus to let the experimenter determine the condition. Before each session began, the Propeller detected the position of the switch and implemented the appropriate experimental condition.

One major finding of this experiment was that the sexual conditioning procedure was much more successful with the parent-reared birds than with the human-reared birds. Figure 8.8 shows the average duration, in seconds, that four human-reared and four parent-reared birds spent within 20 cm of the conditioning light in the eight, 5-minute sessions after the conditioning procedure. These data were taken from the "Total Duration" column of the PEC's standard data spreadsheet. This was a surprising finding, given the well-documented ability of this general procedure to affect behavior in birds (Burns–Cusato, Cusato, and Daniel, 2005; Domjan, Blesbois, and Williams, 1998; Domjan, Lyons, North, and Bruell, 1986). However, this finding was taken as an indicator that sexual conditioning is not an appropriate method to change behavior in human-reared birds.

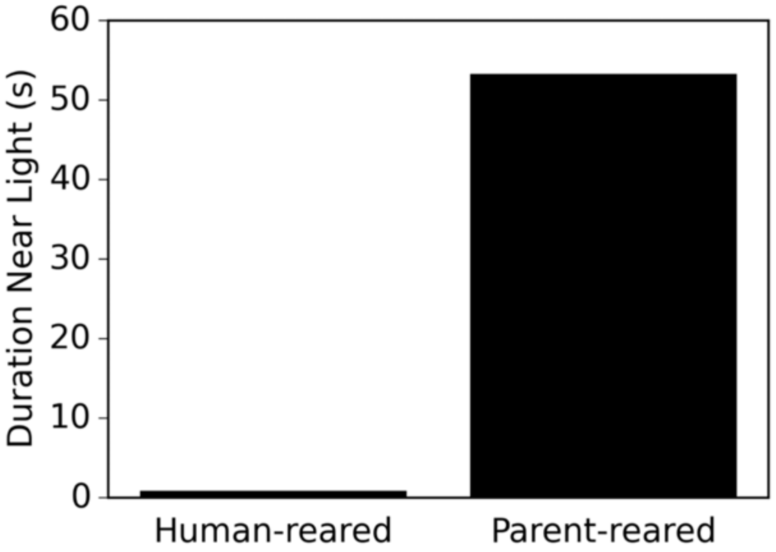

Figure 8.8: Sexual conditioning in pigeons (adapted from Varnon, 2013).

Inexpensive Conditioning and Learning Laboratory

Another early use of the PEC was to create a teaching laboratory for principles of learning and behavior (Varnon and Abramson, 2013). The paper cited here describes how to use the PEC in the classroom and walks the reader through use of 20 programs suitable for a laboratory course in learning including programs on habituation, classical conditioning, schedules of reinforcement, chaining, and operant discrimination. For each program, the paper provides detailed information about how the program works, as well as how to change experimental parameters, defined in the constant section, to adjust the way the experiment is conducted. For example, adjusting experimental parameters in the classical conditioning program can alter the program to allow for trace conditioning, delay conditioning, simultaneous conditioning, or backward conditioning procedures. In addition to being excellent for laboratory exercises, the programs are also suitable for research experiments. As the paper provides instructional information on the use of the programs, it is also helpful for beginners learning how to use the PEC, and for learning how to modify programs for custom experiments. The programs described in the paper are also available at CAVarnon.com/sample-experiments.

Fixed-Interval Schedules of Reinforcement in Horses

The PEC has been used to study the ability of horses (*Equus ferus caballus*) to respond in fixed-interval schedules of reinforcement (Craig et al., 2015). In this experiment, horses could respond by putting their head through a large plastic hoop and breaking a modulated IR beam inside. A variety of conditions were used, but the overall procedure used a fixed-interval schedule of reinforcement, where the horse received a food reinforcer if it responded in a fixed amount of time after it received the previous reinforcer. This general method investigates if an animal learns to predict the time interval and respond more frequently towards the end of the interval. The food reinforcer was delivered by a modified Super-Feeder (as described in the High Voltage Digital Output section of this chapter). The Propeller also played a 330-Hz, 75% duty cycle square wave tone when the horse broke the IR beam, and a 532-Hz, 50% duty cycle square wave tone when food was delivered. Both tones were created using Experimental Functions's PWM methods.

In addition to saving events using the standard data spreadsheet, the Propeller also conducted several analyses of the horses' responses during the session and saved this on a separate file on the SD card. As this process was complex, a separate cog was launched to analyze the data. When the analyses were complete, the main cog was instructed to write the findings of the analyses to the SD card. The technique used to analyze data in a separate cog, and then save data using the main cog, was similar to that in seen in Figure 8.1. One analysis conducted by the Propeller was to sort the number of responses that occurred during each fixed-interval period into 10, equal-sized bins. This allowed for a comparison between the patterns of responding across different fixed-interval conditions. For example, Figure 8.9 shows the average number of responses of horses experienced in the fixed-interval 60-second (n = 3), fixed-interval 90-second (n = 3), and fixed-interval 180-second (n = 5) procedures. For each horse, the average responses during 10 trials of the final fixed-interval session is included. The general finding, as can be seen in the figure, is that horses learn to predict the time interval, much like other mammals. Note that, regardless of condition, the horses tend to respond less toward the beginning of the interval when food is not available, and more toward the end of the interval when food becomes available again.

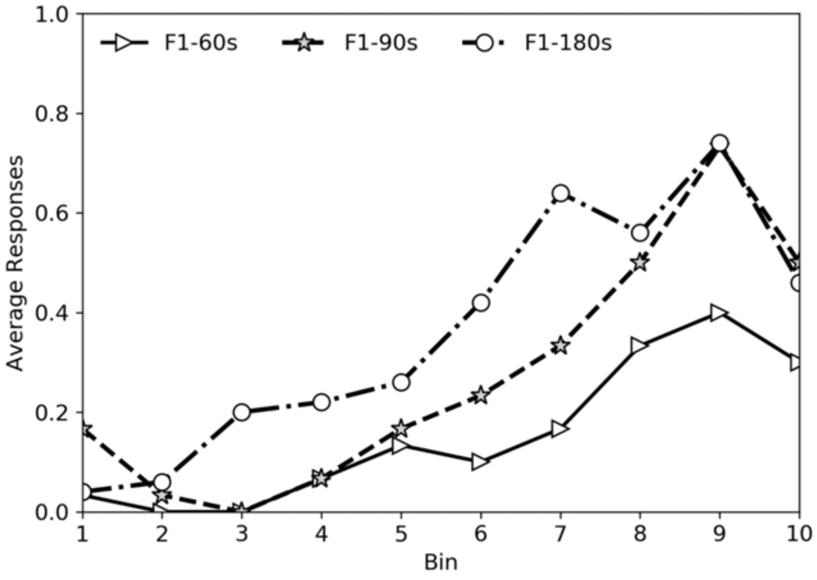

Figure 8.9: Fixed-interval schedule of reinforcement in horses (adapted from Craig et al., 2015).

Foraging Behavior of Wild Squirrels

In addition to laboratory studies, the PEC has also been used to study the foraging behavior of wild eastern fox squirrels (*Sciurus niger*; Phelps and Varnon, 2015). This project used two modified squirrel feeders. The squirrels can obtain food by lifting the lid of the feeder and reaching their head or hands inside. Microswitches were added to detect when the squirrels opened the lid. When the lid was closed, it rested on the microswitches. The platform on which the squirrels sat when accessing the feeder was also modified. A floor plate, resting on microswitches, was added to allow the Propeller to detect when the squirrels were sitting at the feeder. Although the Propeller records every property of these measures, the primary measures of interest were the number of times the squirrels opened the feeder, and the duration the squirrels sat at the feeder. These measures could be quickly obtained from the "Total Occurrences" and "Total Duration" columns of the standard data spreadsheet. The most interesting aspect of this apparatus was that it was used to collect data outdoors. As the Propeller is so small, the entire control unit could be kept in a plastic container near the feeders. The program for the squirrel apparatus is relatively simple. It detects and records input events for the lid and platform of both feeders in a repeat loop. The repeat loop can be terminated by pressing a push-button, allowing for data to be collected continuously until the user wishes to retrieve the data. In practice, the Propeller was turned on early in the morning, and turned off in the evening when retrieving data from the SD card.

The experiments conducted with this apparatus are still in progress. They involve manipulating some aspect of the environment, often adding weights to the feeder lids to increase response cost. The effects of the manipulation can then be observed on the rate the feeder is used. Often increasing the response cost to one feeder results in increasing use of the other feeder.

Aversive Conditioning in Honey Bees

The PEC also has been used to study aversive conditioning in honey bees (*Apis mellifera*; Dinges et al., 2017). In this research, honey bees were placed in shuttle boxes, restricting the movement of the bees but allowing them to move from one compartment in the shuttle box to the other. The location of each bee was detected with a pair of modulated IR beams in the center of each shuttle box (two beams per box). The shuttle boxes sat on shock grids that could be activated by the Propeller through electromechanical relays. In escape experiments, bees could escape shock occurring at regular intervals by quickly crossing from one side of the shuttle box to the other. For these experiments, both shuttle boxes ran independent experiments with each bee. In a second set of punishment experiments, colored paper was placed under the shuttle box to visually divide it into two sides. Bees were shocked when they entered a specific side of the shuttle box. As a control, one bee acted as a master bee, and another as a yoked bee. The master bee was able to avoid shock by moving to the correct side of the apparatus, signaled by the color underneath the shock grid. The yoke bee was not able to avoid shock, and instead was shocked whenever the master bee received shock. The bees also alternated between master and yoke role in some conditions.

Figure 8.10 shows one finding of this series of experiments. In the punishment experiment shown here, bees in the control group (n = 20) were able to avoid shock for two, consecutive, 5-minute sessions by staying on one side of the shuttle box, marked by either blue or yellow color. Bees in the helplessness group (n = 20) were similarly able to avoid shock in the second, 5-minute session, but were not able to avoid shock in the first session. Exposing the helplessness group to unavoidable shock during the first session made it difficult for this group to learn to avoid shock in the second session, a psychological process called learned helplessness. The data in Figure 8.10 show the duration the bees spent on the safe side of the apparatus (away from the shock) in the second, 5-minute session. As expected, the bees in the helplessness group did not avoid shock as well as bees in the control group. These data were taken from the "Total Duration" column of the PEC's standard data spreadsheet.

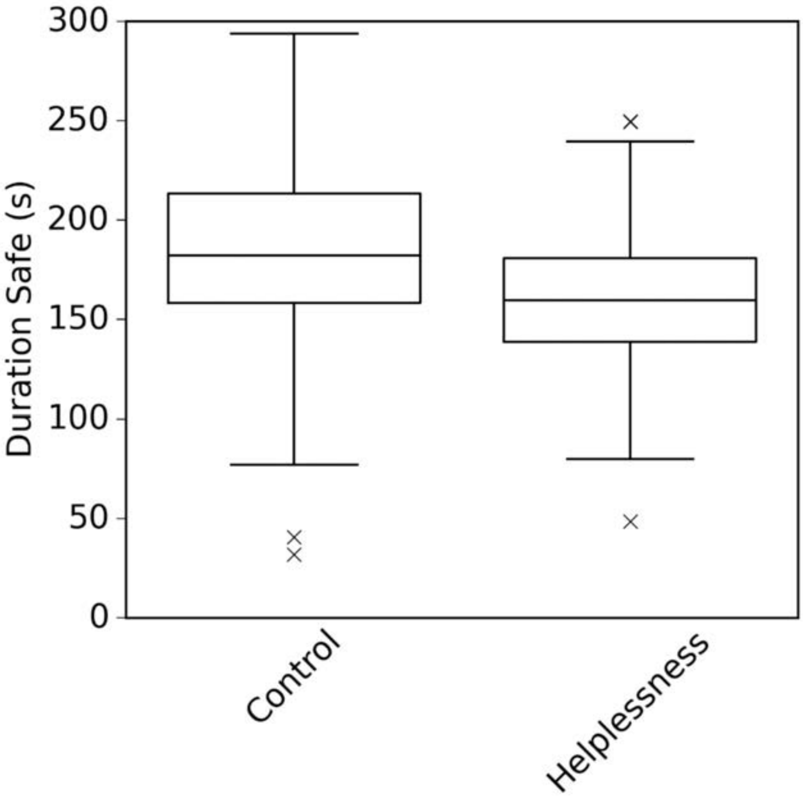

Figure 8.10: Aversive conditioning in honey bees (adapted from Dinges et al., 2017).

The programs for the shuttle box experiments were rather complex in a number of ways. First, the programs had to implement a variety of experimental parameters based on a complex user interface. The state of several input switches was read before the experiment began to determine the experimental condition that each bee would experience. Many of the experiments also required that conditions change over the course of a session. Next, the programs had to create and store data spreadsheets for each bee, for each session. Because the experiment had to be conducted rapidly, there was not time to transfer data from the SD card to the computer after each session. Instead the programs used a settings file containing information about the number of sessions of each condition type that were previously conducted. The data spreadsheet name was modified, in a manner similar to that seen in Figure 8.5, to record the number of sessions (from 1–99) that had been conducted in 31 different groups.

Finally, the location of the bees could not be determined by simple input events. Instead, the programs had to compare the state of both IR beams to determine which side of the shuttle box the bees were on. Because the beams were placed in the center of the shuttle box, it is not possible for the subject to be between them. Therefore, if the bee breaks one beam, but not the other, it must be on the side of the shuttle box with the broken beam. Figure 8.11 shows an example of this logic for one shuttle box. The first conditional evaluates if the bee is on side 1 of the shuttle box. It starts by focusing on the state of IR1, an input event used to detect activation of the IR beam on side 1 of the shuttle box. If IR1 state is onset or on, the program then checks the state of IR2, the IR beam on side 2 of the shuttle box. If IR2 state is offset or off, then the bee must be on side 1. Manual events are then used to record that the bee is on side 1, and a second manual event records that the bee is not on side 2. The second conditional similarly evaluates if the bee is on side 2. Note that this program excerpt only illustrates the logic used to determine which side of the shuttle box the bee is on.

```
REPEAT
    IF IR1.State == Onset OR IR1.State == On
        IF IR2.State == Offset OR IR2.State == Off
            BeeSide := 1
            EXP.Record(Side1.StartManualEvent, Side1.ID, EXP.Time(Start))
            EXP.Record(Side2.StopManualEvent, Side2.ID, EXP.Time(Start))
    IF IR2.State == Onset OR IR2.State == On
        IF IR1.State == Offset OR IR1.State == Off
            BeeSide := 2
            EXP.Record(Side2.StartManualEvent, Side2.ID, EXP.Time(Start))
            EXP.Record(Side1.StopManualEvent, Side1.ID, EXP.Time(Start))
```

Figure 8.11: Shuttle box side detection program excerpt.