7

EXPERIMENTAL FUNCTIONS

The Propeller Experiment Controller's (PEC) Experimental Functions object provides a variety of public methods to fill the diverse needs of behavioral experiments. To accomplish this, many of the methods in Experimental Functions have multiple variations, such as the several variations of the StartExperiment method. Other methods are related and codependent. For example, the SetFrequency method can generate audio frequencies on I/O pins, but it first requires a StartFrequencyGenerator method to be used. Because of the relationships between the methods in Experimental Functions, the methods are grouped together into method families. In the following sections, we will discuss each method family, as well as individual public methods of Experimental Functions. Private methods cannot be used in other programs and thus will not be discussed. The headers for each method represent the method name, with the method parameters in parenthesis.

StartExperiment Methods

Several StartExperiment methods can be used to prepare the Propeller for an experiment. These methods have three primary goals: (1) to start the experiment clock, (2) to generate a random number to use as a seed for future pseudo-random number generation, and (3) to connect to the SD to prepare for recording data using a modified version of the FRSW object (Rokicki and Dummer, 2009). The StartExperiment methods use two cogs. Starting the experiment clock requires one cog. The clock then runs in the background, and can be used for many purposes, such as executing code at specific times and recording temporal variables. The Experimental Event object also requires the experimental

clock for its methods. Preparing the random seed for random number generation temporarily uses a cog. However, after a random number is generated, the cog is then closed and freed for other purposes. Connecting to the SD card also requires a cog. When the StartExperiment method connects to the SD card, it creates a memory file, with a default name of "memory.txt" in the root directory of the SD card. The memory file is used to quickly record very basic information about the experiment while the experiment is in progress. After an experiment ends, the memory file can be used to create a detailed data spreadsheet, with a default name of "data.csv".

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off      = 0
    Ons et   = 1
    On       = 2
    Offset   = 3
    DO       = 0
    CLK      = 1
    DI       = 2
    CS       = 3
OBJ
    EXP:     "Experimental_Functions"
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
```

Figure 7.1: StartExperiment example program.

StartExperiment(DO, CLK, DI, CS). This is the standard StartExperiment method. It prepares a memory file on the SD card named "memory.txt", generates a random number, and then launches the experiment clock. The parameters DO, CLK, DI, and CS refer to the Data Out, Clock, Data In, and Chip Select pins of the SD card. The user must provide the pin numbers of the Propeller's I/O pins connected to the DO, CLK, DI, and CS pins on the SD card. It is important to note that these pins should be used exclusively for the SD card. The StartExperiment method returns to 0 if the SD card is properly mounted; this is generally only used to troubleshoot hardware connections. This method requires two cogs, one to interface with the SD card, while a second cog is used for the experiment clock. Figure 7.1 shows an example of using the StartExperiment method. The SD card connections are defined in the constant section. In this example, the connections match those of the Propeller Platform DNA. Other development boards custom setups may use different pins on the Propeller to connect to the SD card.

StartExperimentCustomMemory(DO, CLK, DI, CS, MemoryFile). This method functions identically to the StartExperiment method, except that the user can provide a custom memory file name. Specifying a custom memory file name can be used for many purposes, including running several sessions in a series. For example, session one might use the name "M1.txt", while session two might use the name "M2.txt". By default, the Propeller searches for the default memory file name to create a data spreadsheet, so this method must be used in conjunction with the appropriate SaveData method (see the SaveData methods section). The parameter MemoryFile, must be the address of a null-terminated string. Note that long file names are not supported. File names can be a maximum of eight characters, plus a three-letter extension. For example, "abc.txt" or "abcdefgh.txt" are valid names, while "abcdefghij.txt" is not a valid name. Figure 7.2 shows an example of using StartExperimentCustomMemory.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off     = 0
    Onset   = 1
    On      = 2
    Offset  = 3
    DO      = 0
    CLK     = 1
    DI      = 2
    CS      = 3
OBJ
    EXP:    "Experimental_Functions"
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS, @FileName)
DAT
    FileName    BYTE    "M1.txt"
```

Figure 7.2: StartExperimentCustomMemory example.

StartExperiment_NoData. This method is used to launch the experiment clock (requires one cog) and generate a random number without connecting to an SD card. The StartExperiment_NoData method is often used when automation is needed, but data records are not required. This method is also useful during initial testing and developing of a program before data recording is added. For advanced users that want to manually code their own customized data output format, this method can be used to launch the experiment clock and generate a random number without preparing a file on the SD card for the PEC's standard data format. The examples in Chapter 6 all

use the StartExperiment_NoData method to illustrate the principles of the Experimental Event object.

Shutdown. The Shutdown method unmounts the SD card and stops the experiment clock. This method is typically used at the end of the program after all data have been saved. This will be discussed in more detail along with the SaveData methods.

Time Methods

StartClock. This method manually starts the experiment clock on a new cog. The clock value increments every millisecond and can run for 24 days before needing to be reset. The system clock also records the number of days that have passed, and the number of milliseconds that have passed in each day. Typically, the StartExperiment methods use the StartClock method to launch the experiment clock. However, the clock can also be launched manually. This should be considered an advanced technique and should not be used in conjunction with the StartExperiment methods. Most users will not use the StartClock method. It is available only to provide more freedom for skilled programmers. We have not used this method in practice; all our applications use one of the StartExperiment methods.

StopClock. This method manually stops the experiment clock cog. Typically, the experiment clock is stopped using the StopExperiment methods. However, advanced users can also manually stop the clock. Like the StartClock method, this method is primarily provided to be comprehensive.

ClockID. This method returns the memory address of the experiment clock. The sole purpose of this method is to provide the address of the experiment clock to the Experimental Event object. The Experimental Event object can then use the system clock value found at the address to debounce inputs and record event duration. Chapter 6 shows several examples of using the ClockID method in Experimental Event objects declaration statements.

SetClock(NewValue). The SetClock method changes the experiment clock value to the millisecond parameter, NewValue. This method can be used in the rare scenario where the clock may run for over 24 days. It can also be used to deliberately set the experiment clock to a specified value, such as one representing the current time of day. The example in Figure 7.3 uses the SetClock method to change the experiment clock value to 28,800,000 milliseconds, or 8 hours. If the Propeller is programmed at 8:00 AM, the experiment clock will match the current time of day.

Time(Event). The Time method is a powerful technique used to execute code at specific times and record temporal variables. It is heavily inspired by the Walter and Palya experiment controller's TIME function (Palya and Walter, 1993; Walter and Palya, 1984). The PEC's Time method returns the time, in

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off      = 0
    Onset    = 1
    On       = 2
    Offset   = 3
    DO       = 0
    CLK      = 1
    DI       = 2
    CS       = 3
OBJ
    EXP:     "Experimental_Functions"
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    EXP.SetClock(28_800_000)
```

Figure 7.3: SetClock example program.

milliseconds, since the millisecond parameter, Event. Time(Event), can therefore be read as *time since event*. The Time method literally subtracts the value of Event from the current clock value. If the statement Time(0) is used, it returns the current value of the experiment clock (clock –0), and can thus be thought of as the *time now*.

Figure 7.4 shows a program that uses the Time method to turn on a feeder for 5 seconds every time a lever is pressed. The program uses a similar constant block as previous examples, except now a constant, FeederDuration, is included to represent the 5-second (5,000 ms) duration that the feeder will be activated after each lever-press. A VAR block is also included with one long-sized variable, FeederStart. This variable will be used to note the time that the feeder is activated. Since FeederStart will be used to record a time value, it should be long-sized to match the long-sized value of the experiment clock. During the main repeat loop of the program, if a lever onset is detected, the feeder is turned on and the current time (EXP.Time(0)) is saved as FeederStart. The program now knows exactly when the feeder was turned on. A second conditional in the repeat loop evaluates how much time has passed since FeederStart (EXP.Time (FeederStart)). If the amount of time that has passed since FeederStart is greater than FeederDuration, the feeder will be turned off. Thus, the feeder will activate for 5 seconds, every time the lever is pressed.

Figure 7.5 modifies the previous example further, resulting in a program that runs for 5 minutes. A new constant, SessionDuration, is added to represent the desired session duration. All time values are provided in milliseconds, so the 5-minute session duration is written as 300,000 ms. A new variable, Start, is also

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off          = 0
    Onset        = 1
    On           = 2
    Offset       = 3
    LeverPin   = 5
    FeederPin = 7
    FeederDuration = 5_000
VAR
    LONG    FeederStart
OBJ
    EXP:     "Experimental_Functions"
    Lever:   "Experimental_Event"
    Feeder:  "Experimental_Event"
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            Feeder.TurnOn
            FeederStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            Feeder.TurnOff
```

Figure 7.4: Time example program 1.

added to record the start time of the experiment. Immediately before the main repeat loop begins, Start is set to the current time. This ensures that the program can always refer to an accurate start time. The experiment clock may not be an accurate representation of the time an experiment starts, because several lines of code occur between the StartExperiment method that starts the experiment clock, and the repeat loop, where the experiment actually begins. The repeat loop itself has also been modified to terminate when the time since Start is greater than SessionDuration. This will cause the repeat loop to end after 5 minutes. After the repeat loop ends, the light is turned off, if it was on at the end of the loop.

Day. This method returns the current day, starting with day 1. The experiment clock automatically increments the day value every 86,400,000 milliseconds. The Day method can be used in experiments or automation projects where the Propeller runs for multiple days at a time.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off          = 0
    Onset        = 1
    On           = 2
    Offset       = 3
    LeverPin     = 5
    FeederPin    = 7
    FeederDuration = 5_000
    SessionDuration = 300_000
VAR
    LONG   FeederStart
    LONG   Start
OBJ
    EXP:     "Experimental_Functions"
    Lever:   "Experimental_Event"
    Feeder:  "Experimental_Event"
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    Start := EXP.Time(Start)
    REPEAT UNTIL EXP.Time(Start) > SessionDuration
        Lever.Detect
        IF Lever.State == Onset
            Feeder.TurnOn
            FeederStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            Feeder.TurnOff
    Feeder.TurnOff
```

Figure 7.5: Time example program 2.

DayTime. This method returns the time during the day, in milliseconds. The value is reset automatically each day. During the first day, this instruction is equivalent to Time(0). The DayTime method can be used to easily report the current time of day in experiments or automation projects where the Propeller runs for multiple days at a time. For precise control of timing, the Time method is superior.

TimeToMilliseconds(TimeStringAddress). The TimeToMilliseconds method converts a specifically formatted string to an integer, in milliseconds. It is one of several methods that converts between millisecond-time and a string.

The parameter TimeStringAddress should be the address of the string. The string should exactly follow the format "HH:MM:SS" for a 24-hour clock or "HH:MM:SS _M" for a 12-hour clock. Providing the string "02:00:00 PM" or "14:00:00", for example, would return 49,600,000 ms. This method, along with other time conversion methods, can be used to easily read and set the experiment clock with respect to the time of day.

MillisecondsTo12HourTime(Milliseconds). This method transforms a value, in milliseconds, to a string time value in the format "HH:MM:SS _M" for a 12-hour clock (AM/PM clock). It returns the address of a string stored in Experimental Functions. For example, providing the integer 49,600,000 would return the string "02:00:00 PM". This method, along with other time conversion methods, can be used to easily read and set the experiment clock with respect to the time of day.

MillisecondsTo24HourTime(Milliseconds). This method transforms a value, in milliseconds, to a string time value in the format "HH:MM:SS" for a standard 24-hour clock (military time). It returns the address of a string stored in Experimental Functions. For example, providing the integer 49,600,000 would return the string "12:00:00". This method, along with other time conversion methods, can be used to easily read and set the experiment clock with respect to the time of day.

Pause(Duration). The Pause method pauses the cog that calls the method for the provided duration, in milliseconds. No code will run during this pause. However, other cogs are free to continue operation. The Pause method is fairly accurate; however, some inaccuracy may accumulate after repeated pauses. For example, a 1-millisecond error may occur after five consecutive 5-minute pauses. A single 25-minute pause is less likely to create this error. If a repeated pause is needed in a loop, SyncPause may be more accurate. SyncPause will consider the time it takes to run the code in a loop, Pause will not.

Figure 7.6 shows an example of using the Pause method in a separate cog. This program is similar to that in Figure 7.4, but it also uses a separate cog to run the ShockPulser method, which delivers a 30-second shock every 30 seconds. The ShockPulser method is launched by the Main method. Inside the ShockPulser method, the new cog first uses the DeclareOutput method to declare the Shock event as an output event on the pin, ShockPin. This enables the cog to control the ShockPin. If the first cog used the DeclareOutput method to declare the Shock event as an output event, the second cog would not be able to control the ShockPin. While the session is in progress, the ShockPulser method turns on shock, pauses for 30 seconds, turns off shock, then pauses for 30 seconds again. The cog running the ShockPulser method cannot do anything during the pause. However, since the ShockPulser method runs in a separate cog, the Main method can operate without pausing. To reduce figure size, constants are provided in list form.

```
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, LeverPin,     #7, FeederPin,      #12, ShockPin
    #5_000, FeederDuration,               #30_000, ShockDuration
VAR
    LONG   FeederStart
    LONG   ShockCogStack[100]
OBJ
    EXP:     "Experimental_Functions"
    Lever:   "Experimental_Event"
    Feeder:  "Experimental_Event"
    Shock:   "Experimental_Event"
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    CogNew(ShockPulser, @ShockCogStack)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            Feeder.TurnOn
            FeederStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            Feeder.TurnOff
PUB ShockPulser
    Shock.DeclareOutput(ShockPin, EXP.ClockID)
    REPEAT
        Shock.TurnOn
        EXP.Pause(ShockDuration)
        Shock.TurnOff
        EXP.Pause(ShockDuration)
```

Figure 7.6: Pause example program.

SyncPause(Duration). The SyncPause method functions similarly to the Pause method, except that it is optimized to create a highly precise, synchronized pause in a repeat loop. SyncPause is more accurate in a repeat loop because it compensates for the duration required to execute the preceding code in the repeat loop. This removes any small cumulative error that may occur after multiple pauses.

To use a synchronized pause, use the StartSync method immediately before a repeat loop. Then, the SyncPause method can be used at the end of the repeat loop. Note that the value created by the StartSync method is shared among cogs. This means that a SyncPause can only be used in one cog at a time. Errors may occur if multiple cogs use the SyncPause method simultaneously. The Propeller Manual (Martin, 2011) provides details on creating synchronized pauses with the WAITCNT instruction. Consider the example in Figure 7.7. This example is a revision of the ShockPulser method from Figure 7.6. In this example, a sync point is created using the StartSync method immediately before the repeat loop begins. Inside the repeat loop, shock is turned on, the cog pauses for 20 seconds, then shock is turned off. Next, the cog executes a synchronized pause of 30 seconds. This synchronized pause includes the time taken to turn shock on, pause for 20 seconds, and turn shock off again. The synchronized pause is therefore around 10 seconds and brings the entire repeat loop to a perfect 30 seconds. Without the synchronized pause, small timing errors may accumulate across the experiment, likely only in the range of a few milliseconds.

StartSync. This method marks a point to use for synchronized pauses. Only one sync point can be created, so caution should be used when attempting synchronized pauses in multiple cogs.

```
PUB ShockPulser
    Shock.DeclareOutput(ShockPin, EXP.ClockID)
    Exp.StartSync
    REPEAT
        Shock.TurnOn
        EXP.Pause(20_000)
        Shock.TurnOff
        EXP.SyncPause(30_000)
```

Figure 7.7: SyncPause example program.

Sleep(Duration). This advanced method functions similarly to the Pause method but is designed to put the Propeller in a power-saving mode to conserve battery life. It should only be necessary when the Propeller is powered by battery. The Sleep method puts the cog that uses it, in addition to the experiment clock cog, in a power-saving sleep mode. The Sleep method uses a less accurate pause than the Pause method. Sacrificing temporal accuracy conserves more power. Both the duration of the sleep and the value of the experiment clock after the Sleep method is used may be slightly inaccurate. In practice, this inaccuracy may result only in a few seconds of error after sleeping for several hours. To save the maximum amount of power, any other active cogs also need to be shut down. Any other cogs that were activated, such as those used to run peripheral devices, will still be active and consuming power. As Experimental Func-

tions also uses an additional cog to communicate with an SD card, the SD card cog will need to be manually deactivated during the sleep and reactivated after the sleep using Experimental Functions's SD card methods. Additional power can be saved by changing the Propeller's clock mode before sleeping using the CLKSET instruction (see the Propeller Manual; Martin, 2011). A decrease in clock frequency will save power, but it will further reduce accuracy of the system clock. The Sleep method, manually deactivating and reactivating the SD card cog, and adjusting the Propeller's clock mode are all advanced techniques that should only be considered when conserving power is a major priority.

PulseOutput(Pin, Duration). The PulseOutput method toggles the state of an output I/O pin for the provided duration. If the I/O pin is off, the PulseOutput method turns it on for the duration; if the I/O pin is on, the PulseOutput method turns it off for the duration. Like the Pause method, the PulseOutput suspends all activity on that cog until it is complete and is subject to only minor cumulative errors. The PulseOutput method is convenient for repeatedly toggling the state of an I/O pin, but it does not provide the data benefit as does manually toggling an Experimental Event output event. PulseOutput is better used when data are not required about a specific output. Figure 7.8 shows a revision of the ShockPulser method from Figure 7.6 that uses the PulseOutput method. Inside the repeat loop, the ShockPin is toggled for 30 seconds. Although this method cannot be used to provide data about shock and is not as precise as a method using a synchronized pause, the simplicity can be appealing.

```
PUB ShockPulser
    REPEAT
        EXP.PulseOutput(ShockPin, 30_000)
        EXP.Pause(30_000)
```

Figure 7.8: PulseOutput example program.

Random Number Generation Methods

The PEC is capable of generating random numbers by combining traditional pseudorandom number generation with a random seed. The random seed acts as the source from which all pseudorandom number generation is derived. When the StartExperiment method is used, Parallax's RealRandom object (Gracey, 2007) is launched in a new cog to generate a random seed. After the random seed is generated, the RealRandom cog is closed, freeing that cog for other uses. The random seed can then be used to generate other random numbers. At any time, a new random seed can be generated, temporarily costing one cog.

GenerateRandomSeed. This method is used by StartExperiment to generate a random seed. The random seed can then be used by other methods to produce

random numbers. It can be used to generate a new random seed at any time. However, it is unlikely a new random seed will be required.

Random. This method randomly generates 0 or 1. It can be used to randomly determine if an event occurs, such as randomly determining if a stimulus light becomes blue or red. This method returns a 1 approximately 50 percent of the time. During three tests, the chance that the random method returned a 1 during 1,000,000 uses of the random method was 50.0530, 50.0364, and 50.0233 percent. The example in Figure 7.9 shows the Main method of program that will randomly deliver food or shock after each lever-press. In the main repeat loop, if the lever's state is an onset, the EXP.Random method randomly generates a 0 or 1. If the number is 0, the feeder is turned on. Otherwise, the number must be 1, and shock is turned on. Note that the CON, VAR, and OBJ blocks are omitted to reduce figure size.

```
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    Shock.DeclareOutput(ShockPin, EXP.ClockID)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            IF EXP.Random == 0
                Feeder.TurnOn
                FeederStart := EXP.Time(0)
            ELSE
                Shock.TurnOn
                ShockStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            Feeder.TurnOff
        IF EXP.Time(ShockStart) > ShockDuration
            Shock.TurnOff
```

Figure 7.9: Random example program.

PseudoRandom(Limit). This method randomly generates a 0 or 1 but will not generate a lengthy sequence of 0's or 1's. The parameter, Limit, allows the user to define the maximum number of consecutive 0's or 1's that will be allowed. If a consecutive sequence meets the user-specified limit, the next number will be selected deliberately to break that sequence. This can be used to create pseudo-random sequences of conditions, when some randomization is desired, but the effects of several consecutive conditions is a concern. For example, if the program in Figure 7.9 uses the instruction EXP.Random(5) instead of EXP.Random, the

```
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, LeverPin      #7, FeederPin,  #5_000, FeederDuration
VAR
    LONG   FeederStart
    BYTE   VR
OBJ
    EXP:     "Experimental_Functions"
    Lever:   "Experimental_Event"
    Feeder:  "Experimental_Event"
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    VR := EXP.RandomRange(1, 7)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            IF Lever.Count => VR
                Feeder.TurnOn
                FeederStart := EXP.Time(0)
                VR := EXP.RandomRange(1, 7)
                Lever.SetCount(0)
        IF EXP.Time(FeederStart) > FeederDuration
            Feeder.TurnOff
```

Figure 7.10: RandomRange example program.

program would never provide more than five consecutive food or shock deliveries. As the PseudoRandom method needs to store the random sequence to evaluate it, using the PseudoRandom method to simultaneously determine multiple events (i.e., randomly determining the state of a light and the state of shock) may lead to errors.

RandomRange(Minimum, Maximum). This method generates a random long-sized integer within a provided range. The range is limited to 2,000,000,000. Negative numbers are allowed. The maximum range can be expressed in a number of ways, such as (0, 2000000000), (−1000000000, 999999999), or (−2000000000, 0). The range is limited by the maximum value of a long, 2,147,483,647, but is rounded down to 2,000,000,000 to make the method easier to implement and use. The average repeated samplings of a range are generally close to the average of the

minimum and maximum numbers. During three tests of 1,000,000 uses of RandomRange(0, 100), the average results of each test were 50.028462, 49.982889, and 49.947029. Figure 7.10 shows a variable-ratio (VR) schedule of reinforcement program, where food is delivered approximately every four lever-presses. Before the repeat loop begins, the first value of VR is determined. Then, in the main repeat loop, if the lever's state is an onset and the count matches or exceeds VR, food is delivered. A new VR is also determined and the lever's count is reset. To reduce figure size, constants are provided in list form.

PseudoRandomRange(Minimum, Maximum, Limit). This method generates a random number within a provided range (with a maximum range of 2,000,000,000) but will not randomly generate a lengthy sequence of the same number. If a sequence of random numbers meets the user-specified limit, the next number will be selected deliberately to break that sequence. This can be used to create pseudorandom sequences of conditions. Note that because it uses global variables, multiple simultaneous uses of PseudoRandomRange will interfere with each other. As with the PseudoRandom method, using this method to simultaneously determine multiple events may lead to errors.

Probability(Chance). The Probability method is used to determine the chance that an event occurs, from 0 to 100 percent. This method expects the Chance parameter to be an integer value ranging from 0 to 100. The value returned by the Probability method is more likely to be a 0 as the Chance parameter approaches 0, and more likely to be a 1 as the Chance parameter approaches 100. The data in Table 7.1 show expected probabilities (provided as the Chance parameter), and the obtained probabilities of the Probability method returning a 1 on 100,000

Table 7.1
Results of Probability Test

Expected	Observed	Expected	Observed
0	00.000	50	49.747
5	05.362	55	54.968
10	10.303	60	59.948
15	15.495	65	65.149
20	20.374	70	70.152
25	25.344	75	74.663
30	30.190	80	79.842
35	35.172	85	84.577
40	39.964	90	89.826
45	45.217	95	94.660

```
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            IF EXP.Probability(70) == 1
                Feeder.TurnOn
                FeederStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            Feeder.TurnOff
```

Figure 7.11: Probability example program.

uses of the method. For each expected probability, the method was run 100,000 times, and then the results were summed and divided by 1,000 to create a percent number with three decimal places. Figure 7.11 shows the Main method of a program similar to that in Figure 7.10, except that food is delivered probabilistically for each lever-press.

Signal Generation Methods

Experimental Functions can use two sets of methods, the Frequency Generator methods and the Pulse-Width Modulation methods, to produce square wave signals on any I/O pin. Square waves are created when an I/O pin set to output mode is rapidly activated and deactivated, causing an oscillation of current to be produced. The frequency that the pin is toggled corresponds to the frequency of the wave. The duration that the pin is active during one wavelength is known as the pulse-width, or duty cycle. A standard square wave has a duty cycle of 50%. That means the pin is on for 50% of the wavelength. Experimental Functions provides methods that can modulate both the frequency and the pulse-width of the square waves.

Figure 7.12 shows two square waves. The x-axis represents time, while the y-axis represents signal strength. The lower points on the waves correspond to an I/O pin being off, and the higher points on the waves correspond to the I/O pin being on. Note that the square waves are always at one strength extreme or another. The Propeller's I/O pins are digital, so no intermediate (analog) states are possible. The top square wave has an equal duration in the high and low states, and therefore has a duty cycle of 50 percent. The duty cycle of the bottom wave varies, generally being larger on the left, and smaller on the right.

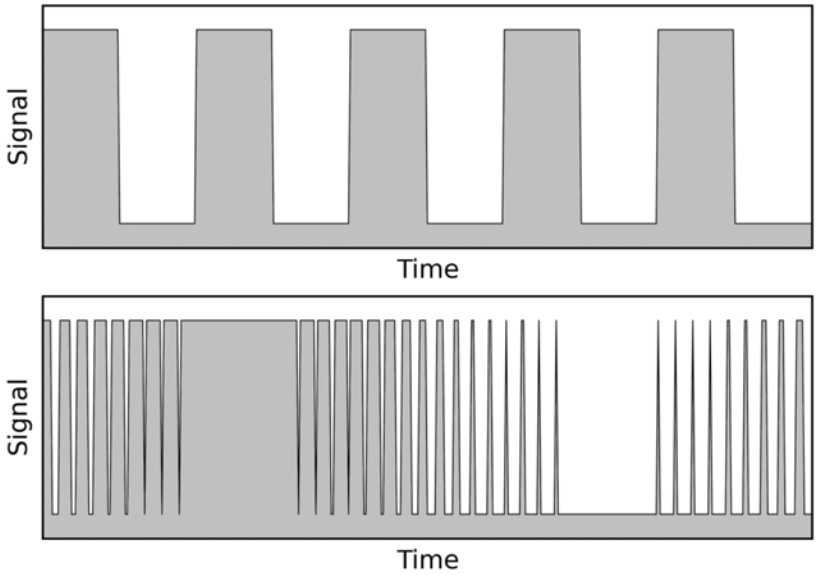

Figure 7.12: Square waves.

The Frequency Generator methods can be used to generate up to two square waves using any I/O pin. These methods result in highly accurate frequencies with a fixed pulse-width of 50%. Typically, these methods are used to generate tones. A frequency generated on a speaker in the range of 20 Hz to 20 kHz will be audible to humans. Other species have different hearing ranges. Lower frequencies can also be used to pulse lights and other devices.

The Frequency Generator methods make use of the cog's counter registers. This allows the frequency generators to work in the background without using an additional cog. However, the counters should not be used for other purposes. See the Propeller Manual (Martin, 2011) for more details on the Propeller's counters. Once the frequencies are set, the designated I/O pin(s) will pulse at that frequency without delaying other tasks. In addition to generating frequencies on the primary pin, inverted frequencies can be generated on another pin. The inverted pin will always be off when the primary pin is on, and vice versa. At high frequencies, the inversion will not be apparent. This can be used to generate the same tone on two audio channels with one frequency generator. At slower frequencies, the inversion is obvious and can be used to do things such as blinking two LEDs in alternation. If the frequency is 0, the inverted pin will always be on. For some applications, such as audio signals, this is not noticeable. The use of inverted pins also increases the number of I/O pins that can be used, from two

to four pins. However, only two independent frequencies can be created with the Frequency Generator methods. Additional frequencies can be generated using one or multiple instances of the EXP_FrequencyGenerator object. This object uses the same methods as Experimental Functions, but each generator will need to be started in a new cog.

The Pulse-Width Modulation (PWM) methods can be used to generate up to 32 square waves using any I/O pin and a modified version of Parallax's PWM object (Schwabe, 2009). These methods allow more frequencies to be generated and allow the pulse-width to be adjusted. However, the signals generated at high frequencies are not accurate. The PWM methods start to lose accuracy at around 2 kHz, and by 4 kHz the accuracy loss is very noticeable. Use the Frequency Generator methods when accuracy of higher frequencies is required. PWM can also be used to generate tones. The pulse-width will affect the timbre of the tone, while pitch is determined by the frequency.

Another major use of PWM is to emulate analog voltage. This technique can be used to decrease the activity a device, such as an LED or motor, by providing a discontinuous source of power. For example, an LED powered by a 50% duty cycle PWM signal will appear dimmer than an LED powered by a 90% duty cycle PWM signal, as the LED powered by a 50% signal is only receiving power 50% of the time. We do not perceive a flicker as the LED is constantly toggled during a PWM signal. Instead we perceive a dimmer light. The point at which we do not perceive individual stimuli is called the flicker fusion threshold. The flicker fusion threshold varies greatly among species and is about 60 Hz for humans. The default frequency of the PWM signals is 1 kHz; this should be well above the flicker fusion threshold for any species. Similarly, PWM can also be used to make motors appear to run slower by rapidly toggling power. In this case, the duty cycle will correspond to the motor speed. Some remaining current and inertia may cause the motor to continue moving during the off portion of the wave. The optimal frequency of the PWM signal depends on the motor. Although 5 kHz should work well for most motors, some experimentation may be useful to find the best frequency. It is also important to consult the motor's datasheet.

The PWM methods can also be used to control **servo motors**. Servos are motors with built-in position detection circuitry. Servos expect to receive a signal every 2000 µs. The duration, or width, of the signal determines the servo's position. Generally, a pulse-width of 1500 µs causes the servo to move to the center position, while pulse-widths of 1000 and 2000 µs cause the servo to move to the extreme positions.

The PWM methods launch a PWM generator, based on Parallax's PWM object (Schwabe, 2009), in a new cog. The generator will not interfere with other cogs. Once the PWM generator is running, a signal can be generated on any I/O pin. The frequency and duty cycle of any of the 32 signals can be modified

independently. Once the frequencies and duty cycles are set, the designated I/O pin(s) will generate those signals without delaying other tasks.

Experimental Functions also provides constants to easily reference musical notes. These constants can be referenced in a program using the notation EXP#NoteName, where NoteName is one of the note constants that can be seen in the Experimental Functions file. This assumes that Experimental Functions has been imported as EXP in the object section of the program. For example, EXP#E4 refers to the note E4, or 330 Hz. The constants can be used with either the Frequency Generator or the PWM methods. The frequencies of the notes are rounded to the nearest hertz.

StartFrequencyGenerator(Pin, InvertedPin). This method starts a frequency generator on a user-provided I/O pin. Optionally, it will generate an inverted frequency on the inverted pin. Provide –1 as the InvertedPin parameter if an inverted frequency is not desired. The generator works by using the counter registers of the cog that called the StartFrequencyGenerator method. A second generator can also be started in that cog. Simply call the method a second time with a new pin to start a new generator. No frequency will be generated until a specific frequency is provided with the SetFrequency method.

SetFrequency(Pin, Frequency). This method sets a frequency, in hertz, on an I/O pin. The user must first start a generator on the I/O pin using the StartFrequencyGenerator method. If an inverted pin was specified in the StartFrequencyGenerator method, an inverted frequency will also be generated on the inverted pin. Setting the frequency to 0 effectively turns off the frequency. Figure 7.13 shows a program that plays a 5-second tone on each lever-press. Note that this program uses a manual event to record information about the tone. The manual event is started immediately before the SetFrequency method sets the frequency to the musical note C4 (262 Hz). The tone manual event is stopped immediately after the SetFrequency method turns the frequency off by setting it to 0.

PlayNote(Pin, Note, Duration). This method sets a frequency on a pin for the duration provided. Note that, like PulseOutput, it suspends all activity on that cog until it is complete. This method provides a convenient way to play short tones or melodies.

StopFrequencyGenerator(Pin). This method stops a frequency generator related to the provided pin. The frequency generator will also be stopped on the inverted pin if one was provided in the StartFrequencyGenerator method. Once all generators have been stopped, the counter registers can be used for other purposes.

StartPWM. This method starts a cog that can generate PWM frequencies on any I/O pin. The PWM generator only needs to be started once. A pin number is not required. Once a pin has been used by the PWM cog, it cannot be used as an input until the PWM cog has been stopped with the StopPWM method.

SetPWMFrequency(Pin, Percent, Frequency). This method generates a PWM frequency on an I/O pin at the provided frequency, in hertz, with the provided

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off           = 0
    Onset         = 1
    On            = 2
    Offset        = 3
    LeverPin      = 5
    SpeakerPin    = 7
    ToneDuration = 5_000
VAR
    LONG   ToneStart
OBJ
    EXP:    "Experimental_Functions"
    Lever:  "Experimental_Event"
    Tone:   "Experimental_Event"
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Tone.DeclareManualEvent(EXP.ClockID)
    EXP.StartFrequencyGenerator(SpeakerPin, –1)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            Tone.StartManualEvent
            EXP.SetFrequency(SpeakerPin, EXP#C4)
            ToneStart := EXP.Time(0)
        IF EXP.Time(ToneStart) > ToneDuration
            EXP.SetFrequency(SpeakerPin, 0)
            Tone.StopManualEvent
```

Figure 7.13: FrequencyGenerator example program.

duty cycle, in percent. After a PWM frequency is set on an I/O pin, no additional commands are needed from the user. The signal will remain until the user changes it or stops the signal generator cog. Figure 7.14 shows a program where a 500 Hz shock is constantly delivered at 75% duty cycle. Pressing a lever decreases the intensity of the shock from 75% to 25% duty cycle for a 5-second period.

SetPWM(Pin, Percent). This method sets the duty cycle, in percent, on a pin with a frequency of 1 kHz. This method is a simplified form of the SetPWMFrequency method. SetPWM is provided for ease of use and to maintain consistency with previous versions of Experimental Functions.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off          = 0
    Onset        = 1
    On           = 2
    Offset       = 3
    LeverPin     = 5
    ShockPin     = 7
    ReliefDuration = 5_000
VAR
    LONG ReliefStart
OBJ
    EXP:    "Experimental_Functions"
    Lever:  "Experimental_Event"
    Shock:  "Experimental_Event"
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Shock.DeclareManualEvent(EXP.ClockID)
    EXP.StartPWM
    Shock.StartManualEvent
    EXP.SetPWMFrequency(ShockPin, 75, 500)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            Shock.StopManualEvent
            EXP.SetPWMFrequency(ShockPin, 25, 500)
            ReliefStart := EXP.Time(0)
        IF EXP.Time(ReliefStart) > ReliefDuration
            EXP.SetPWMFrequency(ShockPin, 75, 500)
            Shock.StartManualEvent
```

Figure 7.14: PWM example program.

Servo(Pin, PulseWidth). This method creates a PWM signal for a servo with the provided pulse-width, in microseconds. For servos, the minimum pulse is around 1000 µs and the maximum pulse is around 2000 µs. A pulse of 1500 µs typically brings servos to a center position. To use this method, start the PWM cog with the StartPWM method, then provide a pulse width in microseconds. Once the signal has been set, the servo will continually receive that signal. No additional commands are needed to maintain the servo's behavior. The example in Figure 7.15 is very similar to that of Figure 7.13, except that a servo-actuated feeder is activated instead of

a tone on every lever-press. Servo-actuated feeders may include modern versions of the classic pigeon hopper feeder or the rat dipper feeder, where the food or liquid is temporarily brought into an animal's reach for a short duration. In the program, before the main repeat loop begins, the servo is put into a position that corresponds

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    Off             = 0
    Onset           = 1
    On              = 2
    Offset          = 3
    LeverPin        = 5
    FeederPin       = 7
    FeederDuration = 5_000
VAR
    LONG   FeederStart
OBJ
    EXP:    "Experimental_Functions"
    Lever:  "Experimental_Event"
    Feeder: "Experimental_Event"
PUB Main
    EXP.StartExperiment_NoData
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareManualEvent(EXP.ClockID)
    EXP.StartPWM
    EXP.Servo(FeederPin, 1200)
    REPEAT
        Lever.Detect
        IF Lever.State == Onset
            Feeder.StartManualEvent
            EXP.SetFrequency(FeederPin, 1800)
            FeederStart := EXP.Time(0)
        IF EXP.Time(FeederStart) > FeederDuration
            EXP.Servo(FeederPin, 1200)
            Feeder.StopManualEvent
```

Figure 7.15: Servo example program.

with a 1200 µs signal. In practice, this might be the retracted position where the food is not accessible to the animal. On each lever onset, the servo is changed to the 1800 µs position that could correspond to an extended position where the animal can reach the food. The servo returns to the 1200 µs position after 5 seconds.

StopPWM. This method stops the PWM cog. This frees the cog and allows all I/O pins used by that cog to be used again.

Number and String Conversion Methods

Experimental Functions provides two methods to convert between integers and strings used to represent integers. These methods are provided to make Experimental Functions a comprehensive toolkit for many Propeller projects. They are especially useful in conjunction with other techniques to display information on the Parallax Serial Terminal, an LCD screen, or a computer monitor (see Chapters 3 and 8 for more details on these applications). Note that these methods are used to convert multi-digit numbers to strings, and vice versa. Single-digit numbers can be easily converted into characters by adding "0" (decimal 48) to the number. Likewise, subtracting "0" from a number character will result in the actual integer value. For example, 5 + "0" is really 5 + 48, and equals 53, or the character "5". See the character charts in Appendix C for a list of characters and their decimal and hexadecimal forms.

ToStr(Number). This method converts an integer to a string. For example, the integer 136 could be converted to the string "136". After converting the number to a string, the string is stored in Experimental Functions's variable space. The ToStr method returns the address to this string so it can be used by other parts of a program. The user should provide an integer as the Number parameter.

ToDec(StringAddress). This method converts a string representing an integer to a decimal (base 10) integer. For example, the string "258" could be converted to the integer 258. The user should provide the address of the string located in memory as the StringAddress parameter. Commas and other symbols should not be included in the string. Figure 7.16 shows an example of integer/string conversion. First, a word-sized variable, NumberString is created to store a string address, and a long-sized variable, NumberInteger, is created to store integers. In the Main method, NumberString is first set to the string "1234". Recall that the STRING instruction actually returns the address of the string in memory. Thus, NumberString now contains the memory address of the string "1234". NumberInteger is then assigned to be the integer conversion of NumberString and is therefore the integer 1234. NumberInteger is then doubled, and NumberString is set to the string conversion of the new value for NumberInteger. The program results in NumberString representing the address to the string "2468".

Array Methods

Experimental Functions provides a long array that can be used in place of user-generated arrays created in VAR or DAT blocks. Ultimately, the purpose of providing this array is to conserve memory. Experimental Functions uses

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
VAR
    WORD NumberString
    LONG NumberInteger
OBJ
    EXP: "Experimental_Functions"
PUB Main
    NumberString := STRING("1234")
    NumberInteger := EXP ToDec(NumberString)
    NumberInteger := NumberInteger * 2
    NumberString := EXP.ToStr(NumberInteger)
```

Figure 7.16: Integer and string conversion example program.

the array itself when creating the data spreadsheet. Thus, the array is a reserved, but unused space during the bulk of an experiment program. The default array length is 2,000 longs. Each long is initialized to be –1. The array length can be adjusted by modifying the MaximumInstances constant in Experimental Functions. The array length is twice the constant MaximumInstances. Altering this constant also changes the maximum instances of a single event that Experimental Functions can save to the data spreadsheet. It should be modified with caution. Because Experimental Functions uses the same array space, the array must be cleared before using the SaveData methods.

Array. This method returns the address of the array in Experimental Functions. The address can be used to read or modify the contents as with any other long array. For example, the statement LONG[EXP.Array][0] := 1 assigns the first long in the array to be the integer 1, and the statement X := LONG[EXP. Array][199] assigns X to equal the 200th long in the array.

ClearArray. This method resets the array after it has been used. All long-sized data in the array will become –1. The array must be reset before using the SaveData methods.

SD Card Methods

Experimental Functions provides several methods to allow the user to manually read and write files on the SD card. These methods use the same techniques and cog that Experimental Functions normally uses to automatically write data to the SD card. The SD Card methods are best used by only experienced programmers with a detailed understanding of the workflow of the PEC.

SDMountCard(DO, CLK, DI, CS). This method mounts the SD card in a new cog. It returns 0 if successful. Once the SD card is mounted, files can be read and

written on the card. The SD card will need to be unmounted in order to safely transfer the files to a computer. The StartExperiment method normally mounts the SD card, and the StopExperiment method normally unmounts the SD card. The SD card can be manually used before StartExperiment and after StopExperiment. The StartExperiment_NoData method also allows the users to manually mount the SD card. As with the StartExperiment methods, the parameters DO, CLK, DI, and CS refer the Data Out, Clock, Data In, and Chip Select pins of the SD card.

SDUnmountCard. This method unmounts the SD card, freeing the SD card. Any opened files are also closed before the card is unmounted.

SDFileOpen(Name, Mode). The SDFileOpen method opens or creates a file on the SD card for reading, writing, appending, or deleting. Only one file can be opened at a time, so this method also closes any previously opened files. The SDFileOpen method returns 0 if successful. A string address should be provided for the parameter, Name. File names can be a maximum of eight characters, plus a three-letter extension. The mode parameter determines the actions that can be performed with the file. Provide the character "r" to open the file for reading; "w" to open the file for writing; "a" to open the file for appending; and "d" to delete the file. The writing and appending mode both allow information to be written to the file, however, writing mode overwrites the existing file while appending mode adds to the end of the file. If a file matching the file name is not already present, writing and appending mode will create a new file. The example in Figure 7.17 mounts the SD card, opens the file "File1.txt" for reading, opens "File2.txt" for writing, opens "File3.txt" for appending, deletes "File4.txt", then unmounts the SD card. Notice that the SDFileClose method is not needed; each time a new file is opened, the previous file is closed. The SDUnmount file also closes any open files.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    #0, DO,   #1, CLK,    #2, DI,    #3, CS
OBJ
    EXP: "Experimental_Functions"
PUB Main
    EXP.SDMountCard(DO, CLK, DI, CS)
    EXP.SDFileOpen(string("File1.txt"), "r")
    EXP.SDFileOpen(string("File2.txt"), "w")
    EXP.SDFileOpen(string("File3.txt"), "a")
    EXP.SDFileOpen(string("File4.txt"), "d")
    EXP.SDUnmountCard
```

Figure 7.17: SDFileOpen example program.

SDFileClose. This method closes the currently opened file. Closing the file is required to save any of the contents that may have been modified in write or append mode.

SDReadCharacter. This method reads a single byte character from a file previously opened in reading mode. The method starts by returning the first character of the file. The next time the SDReadCharacter method is used, the second character will be returned. The method increments through the file, returning each character. When the end of the file is reached, a –1 will be returned. Figure 7.18 shows an example of the SDReadCharacter method. A byte array, Character, is created with 6 bytes. The first 5 bytes of Character are assigned to be the first 5 bytes found in "File1.txt". If the file contained the line "Hello world!", then the byte array, Character, could be interpreted as the string "Hello". As Character[5] was not assigned, it remains a 0, and is thus suitable for the end of a null-terminated string.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    #0, DO,   #1, CLK,  #2, DI,   #3, CS
VAR
    BYTE Character[6]
OBJ
    EXP: "Experimental_Functions"
PUB Main
    EXP.SDMountCard(DO, CLK, DI, CS)
    EXP.SDFileOpen(string("File1.txt"), "r")
    Character[0] := EXP.SDReadCharacter
    Character[1] := EXP.SDReadCharacter
    Character[2] := EXP.SDReadCharacter
    Character[3] := EXP.SDReadCharacter
    Character[4] := EXP.SDReadCharacter
    EXP.SDUnmountCard
```

Figure 7.18: SDReadCharacter example program.

SDReadSequence(ResultArrayAddress, ResultArrayLength, Delimiter). This method reads a sequence of characters from a file opened in reading mode. The sequence will be stored in a results array that has been already created by the user, and the address of that array will be returned. The SDReadSequence method requires the user to supply the memory address of the results array, using the @ operator, as well as the length of the results array. The results array must be large enough to accommodate the sequence that will be read. The method stops reading from the file at the end of the sequence, noted by a predefined delimiter character, such as a comma or period. The user must provide the delimiter character as

the parameter, Delimiter. Any unused values in the results array will be set to 0. Similar to the use of SDReadCharacter, repeatedly using the SDReadSequence method will allow the user to read through each sequence in the file.

The example shown in Figure 7.19 uses the SDReadSequence method to read a sequence in "File1.txt". The SDReadSequence method starts readying the file with the first character and stops with the first occurrence of the delimiter character, "/". If the file contains the line "Hello world!/", then the Parallax Serial Terminal will display the string "Hello world!". However, if the delimiter parameter was a space (" "), the Parallax Serial Terminal will only display "Hello". As the SDReadSequence method returns the address of the results array it can be combined with other instructions requiring an array address to reduce lines of code. In this example the EXP.SDReadSequence and PST.Str instructions can be combined into: PST.Str(EXP.SDReadSequence(@SequenceArray, 100, "/"))

SDWriteCharacter(Character). This method writes a single character to a file opened for writing or appending. It returns 0 if successful. The example in Figure 7.20 results in a file, "TestFile.txt", being created on the SD containing the line "TEST". Because the file is opened for writing, if the file was already present on the SD card, it will be overwritten.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    #0, DO,   #1, CLK,  #2, DI,   #3, CS
VAR
    BYTE    SequenceArray[100]
OBJ
    EXP:    "Experimental_Functions"
    PST:    "Parallax Serial Terminal"
PUB Main
    PST.Start(115_200)
    EXP.SDMountCard(DO, CLK, DI, CS)
    EXP.SDFileOpen(string("File1.txt"), "r")
    EXP.SDReadSequence(@SequenceArray, 100, "/")
    PST.Str(@SequenceArray)
    EXP.SDUnmountCard
```

Figure 7.19: SDReadSequence example program.

SDWriteString(StringAddress). This method writes the string at the provided address, StringAddress, to a file opened for writing or appending. This method does not write the string's null-terminator character. If successful, this method returns the number of characters written. The example in Figure 7.21 opens the file "TestFile.txt" for appending. It creates a new line with the characters 13 and

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    #0, DO,   #1, CLK,  #2, DI,   #3, CS
OBJ
    EXP:      "Experimental_Functions"
PUB Main
    EXP.SDMountCard(DO, CLK, DI, CS)
    EXP.SDFileOpen(string("TestFile.txt"), "w")
    EXP.SDWriteCharacter("T")
    EXP.SDWriteCharacter("E")
    EXP.SDWriteCharacter("S")
    EXP.SDWriteCharacter("T")
    EXP.SDUnmountCard
```

Figure 7.20: SDWriteCharacter example program.

10 (see the character charts in Appendix C for more information about special characters). Then, it writes the string "Test number: ". Next, it uses the EXP.ToStr method to convert the integer 123 to a string and writes the result in the file. The program will result in a new line being added to the file containing the line "Test number: 123". If the file was not already present, a new file will be created.

SDWriteSeconds(Milliseconds). This method converts a millisecond integer into a string representing seconds, with a three decimal place fraction. The string is then written to the file currently opened for writing or appending. This method does not write the string's null-terminator character. For example, the instruction EXP.SDWriteSeconds(1234) would write the string "1.234" to the file.

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000
    #0, DO,   #1, CLK,  #2, DI,   #3, CS
OBJ
    EXP:      "Experimental_Functions"
PUB Main
    EXP.SDMountCard(DO, CLK, DI, CS)
    EXP.SDFileOpen(string("TestFile.txt"), "a")
    EXP.SDWriteCharacter(13)
    EXP.SDWriteCharacter(10)
    EXP.SDWriteString(STRING("Test number: "))
    EXP.SDWriteString(EXP.ToStr(123))
    EXP.SDUnmountCard
```

Figure 7.21: SDWriteString example program.

Memory File Methods

Experimental Functions offers a few methods to manually interact with the memory file. These methods are advanced and should only be implemented by users with a good understanding of the standard PEC program workflow.

QuickSaveMemory. This method quickly closes and re-opens the memory file to save the information currently recorded. This method may be useful in experiments with longer sessions, especially when there is potential for the power source to be interrupted.

QuickSaveCustomMemory(MemoryFile). This method functions similarly to QuickSaveMemory, except that it permits custom memory file, provided by the string address parameter MemoryFile, to be used instead of the default "memory.txt" file. This method should be used if StartExperimentCustomMemory is used. Note that long file names are not supported. File names can be a maximum of eight characters, plus a three-letter extension.

CreateMemoryFile(MemoryFile). This method creates a new memory file with a name provided by the string address parameter, MemoryFile. This method can be used to create a memory file for a new session after the data from the first session has been saved. File names can be a maximum of eight characters, plus a three-letter extension.

Record Methods

The Record methods are used to quickly save information to the memory file during the course of an experiment session. These methods require one or multiple Experimental Event objects. During an experiment, each Experimental Event can pass information to Experimental Functions using the Record methods. Experimental Functions then saves this information to the memory file.

Record(State, ID, EventTime). This method is used to record data about input events, output events, and manual data events. The Record method requires a State parameter to record event states. Only onsets and offsets will be recorded. As the Detect, DetectInverted, TurnOn, TurnOff, StartManualEvent, and StopManualEvent methods from Experimental Event all return the event state, they are designed to be nested inside the Record method. This allows for any instruction in Experimental Event that detects or affects an event to be easily recorded. The ID parameter is the ID of the event that will be written in the memory file. When an Experimental Event is declared, it is automatically assigned an event ID. The event ID can be provided to the Record method with Experimental Event's ID method. The EventTime parameter refers to the time that will be recorded if the event state is an onset or an offset. Typically, the current time will be provided.

Figure 7.22 shows a modification of the program first described in Figure 7.5. The program activates a feeder for 5 seconds any time the lever's state is an onset

```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq  = 5_000_000
    Off       = 0
    Onset     = 1
    On        = 2
    Offset    = 3
    DO        = 0
    CLK       = 1
    DI        = 2
    CS        = 3
    LeverPin  = 5
    FeederPin = 7
    FeederDuration = 5_000
    SessionDuration = 300_000
VAR
    LONG   FeederStart
    LONG   Start
OBJ
    EXP:     "Experimental_Functions"
    Lever:   "Experimental_Event"
    Feeder:  "Experimental_Event"
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    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))
```

Figure 7.22: Record example program.

during a 5-minute session. To modify the program to record data, the StartExperiment_NoData method was changed to StartExperiment, and the connections to the SD card were provided as parameters. Then, each Detect, TurnOn, and TurnOff method was wrapped in a Record method. For example, the first line of the main repeat loop was Lever.Detect. The line now becomes EXP.Record(Lever.Detect, Lever.ID, EXP.Time(Start)). When this code runs, the lever's state is detected just as before. The state is also passed to the Record method, along with the lever's ID, and the time since the session start. If an onset or offset was detected, it will be recorded to the memory file. The last two lines of code evaluate if the lever was in the onset or on state when the session ended. If so, an onset was written to the memory file, so an offset needs to be written as well. The last line forces an offset to be recoded using the lever event's ID. This last step is technically optional but forcing the program to write offsets for any ongoing events cleans up the data considerably. Note that nothing happens to the information in the memory file yet. Additional methods will be used to transform the memory file into a data spreadsheet.

RecordRawData(Integer, ID, EventTime). This method saves integer data for raw data events. Like the Record method, the information is saved to the memory file. This method should be used only when some raw integer data have been collected. Figure 7.23 shows a modification of Figure 7.22 where an additional Experimental Event, Pulse, is added, as well as a hypothetical HeartRateSensor object, referred to as HRS. The RecordRawData event records the pulse using the HRS.BMP method on each lever onset. Although the use of the HRS object is just a hypothetical demonstration, this general technique is very useful when recording information from analog or complex devices. To reduce figure size, constants are provided in list form.

SaveData Methods

The SaveData methods are all related to creating the data spreadsheet at the end of an experiment from the information stored in the memory file. These should be used after the experiment is complete to create the data spreadsheet. During this process, the Propeller searches through memory file for information about a specific event. The Propeller is able to store information, about 1,000 instances of the event, in Experimental Functions's memory array (see the Array Methods section). Then, additional information about the event is derived, and detailed information is saved on the data spreadsheet. If more than 1,000 instances of an event occur, only the first 1,000 will be saved to the data spreadsheet. The 1,000-instance limit can be adjusted by changing the constant MaximumInstances in Experimental Functions. This can be done to increase the limit, or to free more memory for other aspects of the program. In practice, the 1,000-instance limit should rarely be a concern.

```
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, LeverPin = 5,             #7, FeederPin
    #5_000, FeederDuration,       #300_000, SessionDuration
VAR
    LONG   FeederStart
    LONG   Start
OBJ
    EXP:     "Experimental_Functions"
    Lever:   "Experimental_Event"
    Feeder:  "Experimental_Event"
    HRS:     "HeartRateSensor"
    Pulse:   "Experimental_Event"
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    Pulse.DeclareRawData(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)
            EXP.RecordRawData(HRS.BPM, Pulse.ID, EXP.Time(Start))
        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))
```

Figure 7.23: RecordRawData example program.

PrepareDataOutput. This method creates the data spreadsheet file named "data.csv". It then adds the headings for each column: Event, Instance, Onset, Offset, Duration, Inter-Event Interval, Total Duration, and Total Occurrences. This method also closes the memory file automatically. The PrepareDataOutput method should be used after the experiment is complete.

PrepareCustomDataOutput(DataFile). This method is similar to the PrepareDataOutput file, except that it uses a custom data spreadsheet name pro-

vided as the string address parameter, DataFile. User specified data spreadsheet names allow for multiple data spreadsheets to be written for one experiment, perhaps splitting the events into two spreadsheets. This could allow a single program to monitor several experimental apparatuses and create a data spreadsheet for each subject. Custom data spreadsheet names also allow for multiple data spreadsheets to be written for multiple consecutive sessions (e.g., "data1.csv", "data2.csv", "data3.csv", etc.). Note that long file names are not supported. File names can be a maximum of eight characters, plus a three-letter extension.

SaveData(ID, Name). This method can be called after the experiment ends, and after the PrepareData method has created a data spreadsheet. The SaveData method searches through the memory file for any event matching the provided ID and saves detailed information in the data spreadsheet using the provided string address parameter, Name. Figure 7.24 builds upon the example from Figure 7.23. Previously, the Record method was added to record information about events in the memory file. Now, the PrepareDataOutput and SaveData methods are added to create the data spreadsheet. The Shutdown method stops the SD card cog (as well as the experiment clock), allowing the SD card containing the finished data spreadsheet to be transferred to the computer. Note that the heart rate data have not been saved yet. To reduce figure size, constants are provided in list form.

SaveCustomData(ID, Name, MemoryFile, DataFile). This method functions similarly to the SaveData method, except that both the memory file and the data spreadsheet name can be customized. File names can be a maximum of eight characters, plus a three-letter extension.

PrepareDataOutputForRawData(DataFile). This method is used to prepare a data spreadsheet for a raw data event. Raw data events are used to record integers at instantaneous points in time and thus do not have an onset, offset, or duration. The column headers for other event types, as created by the PrepareDataOutput and PrepareCustomDataOutput methods, are not appropriate for raw data events. Therefore, the PrepareDataOutputForRawData method uses the headers: Event, Instance, Time, Data, Inter-Event Interval, and Total Occurrences. This method will create a new data spreadsheet, if a data spreadsheet matching the name provided by the string address parameter, DataFile, is not found. However, if a data spreadsheet is found, this method will skip one line, and then add the raw data headers to that file. If adding raw data to an existing data file, the PrepareDataOutputForRawData method should be used after all other data are saved. If you want to save raw data into a separate data file, simply provide a unique file name. File names can be a maximum of eight characters, plus a three-letter extension.

SaveRawData(ID, Name, Memoryfile, Datafile). This method functions similarly to the SaveCustomData method, except that it is used to save information

```
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, LeverPin,                   #7, FeederPin
    #5_000, FeederDuration,    #300_000, SessionDuration
VAR
    LONG   FeederStart
    LONG   Start
OBJ
    EXP:     "Experimental_Functions"
    Lever:   "Experimental_Event"
    Feeder:  "Experimental_Event"
    HRS:     "HeartRateSensor"
    Pulse:   "Experimental_Event"
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    Pulse.DeclareRawData(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)
            EXP.RecordRawData(HRS.BPM, Pulse.ID, EXP.Time(Start))
        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.PrepareDataOutput
    EXP.SaveData(Lever.ID, STRING("Lever"))
    EXP.SaveData(Feeder.ID, STRING("Feeder"))
    EXP.Shutdown
```

Figure 7.24: SaveData example program.

```
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, LeverPin,                   #7, FeederPin
    #5_000, FeederDuration,         #300_000, SessionDuration
VAR
    LONG   FeederStart
    LONG   Start
OBJ
    EXP:    "Experimental_Functions"
    Lever:  "Experimental_Event"
    Feeder: "Experimental_Event"
    HRS:    "HeartRateSensor"
    Pulse:  "Experimental_Event"
PUB Main
    EXP.StartExperiment(DO, CLK, DI, CS)
    Lever.DeclareInput(LeverPin, EXP.ClockID)
    Feeder.DeclareOutput(FeederPin, EXP.ClockID)
    Pulse.DeclareRawData(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)
            EXP.RecordRawData(HRS.BPM, Pulse.ID, EXP.Time(Start))
        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.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
```

Figure 7.25: SaveRawData example program.

about raw data events. Raw data events use a different header system, so they should be saved in a separate data file or saved after all other events. File names can be a maximum of eight characters, plus a three-letter extension. Figure 7.25 adds to the previous example and prepares the data spreadsheet for raw data after the other events have been saved. Then, the heart rate data are saved using the SaveRawData method. Finally, the shutdown method unmounts the SD card and stops the experiment clock. To reduce figure size, constants are provided in list form.

SaveMPCData(ID, OnsetCode, OffsetCode, MemoryFile, Datafile). This method saves data recorded during an experiment in a MedPC style format. The standard MedPC output is minimalistic and is usually sorted by some other program. The SaveMPCData method is provided so that users already having software to sort their MedPC data output can still use the PEC with very few adjustments in how the data are interpreted. The MedPC format often lists events in plain text file with the format time.code, where time is the time the event occurred, and code is some integer code for the event. As the MedPC system is designed to primarily record the onset of an event, onsets and offsets of each event will be saved separately using different codes. For example, after an experiment is complete, lever-press data could be saved using the code EXP.SaveMPCData(Lever.ID, STRING("001"), STRING("002")). In the data file, lever onsets would be listed as time.001 and offsets would be listed as time.002. If the lever was pressed from 10 seconds to 12 seconds, 10000.001 would show the onset, in milliseconds, on one line, while 12000.002 would show the offset on another line. The parameters OnsetCode, OffsetCode, MemoryFile, and DataFile must all be provided as string addresses. File names can be a maximum of eight characters, plus a three-letter extension. Because the MPC format is typically used to create text files sorted by another program, no PrepareDataOutput method is required to generate a spreadsheet and add column headers.

SaveRawMPCData(ID, TimeCode, DataCode, MemoryFile, Datafile). This method functions similarly to the SaveMPCData method, except that it saves integer data from raw data events. The time of measurement and the data collected are saved separately using different codes. Figure 7.26 shows a modification of the previous example that saves MedPC formatted data. The CON, VAR, and OBJ blocks have been removed to reduce figure size. Data for each event are saved using the SaveMPCData and SaveRawMPCData methods. A DAT block is included to store the strings needed by these methods. Notice the @ operator is used to provide the address of these strings. In this example, the default memory file name was used. The data file, however, is saved as a text file instead of a comma-separate value file. This is because the formatting of the MPC output will not benefit from being viewed in a spreadsheet program.

```
PUB Main
  EXP.StartExperiment(DO, CLK, DI, CS)
  Lever.DeclareInput(LeverPin, EXP.ClockID)
  Feeder.DeclareOutput(FeederPin, EXP.ClockID)
  Pulse.DeclareRawData(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)
      EXP.RecordRawData(HRS.BPM, Pulse.ID, EXP.Time(Start))
    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.SaveMPCData(Lever.ID, @L_Onset, @L_Offset, @MemoryFile,@DataFile)
  EXP.SaveMPCData(Feeder.ID, @F_Onset, @F_Offset, @MemoryFile, @DataFile)
  EXP.SaveRawMPCData(Pulse.ID, @H_Time, @H_Data, @MemoryFile, @DataFile)
  EXP.Shutdown
DAT
  L_Onset       BYTE    "001", 0
  L_Offset      BYTE    "002", 0
  F_Onset       BYTE    "003", 0
  F_Onset       BYTE    "004", 0
  H_Time        BYTE    "005", 0
  H_Data        BYTE    "006", 0
  MemoryFile    BYTE    "memory.txt", 0
  DataFile      BYTE    "data.txt", 0
```

Figure 7.26: SaveMPCData and SaveRawMPCData example program. Note that short indentations were used for this program to ensure that the figure fit within a single page. Indentation size does not affect the program's function.