CHAPTER 9

Output Compare

The output compare (OC) peripheral sets the state of an output pin based on the value of a timer. Output compare can be used to generate a single pulse of specified duration or a continuous pulse train. Either mode of operation can generate an interrupt when the value of the output pin changes.

Like most microcontrollers, the PIC32 cannot output an arbitrary analog voltage because it lacks a true digital-to-analog converter (DAC) (see Chapter 16 for details about the PIC32's limited analog output capability). By generating a pulse train, the output compare can be used to generate a time-based analog output. The analog value is proportional to the *duty cycle* of the pulse train: the percentage of the period that the signal is high. Generating a signal whose value is determined by the duty cycle is called "pulse width modulation" (PWM) (see Figure 9.1). High-frequency PWM signals can be low-pass filtered to create a true analog output. PWM signals are also commonly used as input to H-bridge amplifiers that drive motors.

9.1 Overview

The PIC32's five OC peripherals can be configured to operate in seven different modes. In every mode, the module uses either the count of the 16-bit timer Timer2 or Timer3, or the count of the 32-bit timer Timer23, depending on the OC control SFR OCxCON (where x = 1 to 5 refers to the particular output compare module). We call the timer used by an output compare module Timery, where y = 2, 3, or 23. You must configure Timery with its own prescaler and period register, which influences the OC peripheral's behavior.

The OC peripheral's operating modes consist of three "single compare" modes, two "dual compare" modes, and two PWM modes. In the three single compare modes, the Timery count TMRy is compared to the value in the OCx count SFR OCxR. In the "driven high" single compare mode, the OCx output is initially driven low when OCx is enabled, and then transitions to high when TMRy first matches OCxR. In the "driven low" single compare mode, the OCx output is initially driven high and then transitions to low on the first TMRy match. In the "toggle" single compare mode, the OCx output is initially driven low and then toggles on each TMRy match. This toggle mode generates a continuous pulse train.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00009-3

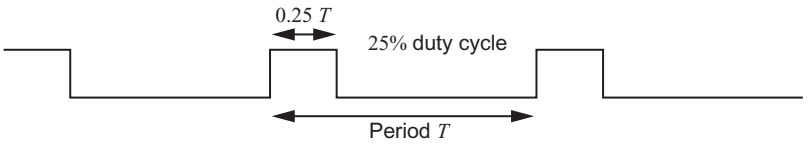

Figure 9.1
A PWM waveform. The duty cycle is the percentage of a period that the signal is high.

In the dual compare modes, TMRy is compared to two values, OCxR and OCxRS. When TMRy matches OCxR the output is driven high, and when it matches OCxRS it is driven low. Depending on a bit in OCxCON, either a single pulse or continuous pulse train is produced.

The two PWM modes create continuous pulse trains. Each pulse begins (is set high) at the rollover of Timery, as set by the period register PRy. The output is then set low when the timer count reaches OCxR. To change the value of OCxR, the user's program may alter the value in OCxRS at any time. This value will then be transferred to OCxR at the beginning of the new time period. The duty cycle of the pulse train, as a percentage, is

```
duty cycle = OCxR/(PRy + 1) x 100%.
```

One of the two PWM modes offers the use of a fault protection input. If chosen, the OCFA input pin (corresponding to OC1 through OC4) or the OCFB input pin (corresponding to OC5) must be high for PWM to operate. If the pin drops to logic low, corresponding to some external fault condition, the PWM output will be high impedance (like an open-drain output, effectively disconnected) until the fault condition is removed and the PWM mode is reset by a write to OCxCON.

9.2 Details

The output compare modules are controlled by the following SFRs. The OCxCON SFRs default to 0x0000 on reset; the OCxR and OCxRS SFR values are unknown after reset.

OCxCON, x = 1 to 5 This output compare control SFR determines the operating mode of OCx.

OCxCON⟨15⟩, or OCxCONbits.ON: Enables and disables the output compare module.
- 1 Output compare enabled.
- 0 Output compare disabled.

OCxCON⟨5⟩, or OCxCONbits.OC32: Determines which timer to use.
- 1 Use the 32-bit timer Timer23.
- 0 Use a 16-bit timer, either Timer2 or Timer3.

OCxCON⟨4⟩, or OCxCONbits.OCFLT: The read-only PWM fault condition status bit. If a fault has occurred you must reset the PWM module by writing to OCxCONbits.OCM (assuming the external fault condition has been removed).

1 PWM fault has occurred.
0 No fault has occurred.

OCxCON⟨3⟩, or OCxCONbits.OCTSEL: This timer select bit chooses the timer used for comparison. If using the 32-bit Timer23, then this bit is ignored.
1 Use Timer3.
0 Use Timer2.

OCxCON⟨2:0⟩, or OCxCONbits.OCM: These three bits determine the operating mode:

0b111 PWM mode with fault pin enabled. OCx is set high on the timer rollover, then set low when the timer value matches OCxR. The SFR OCxRS can be altered at any time, and its value is copied to OCxR at the beginning of the next timer period.[1] The duty cycle of the PWM signal is

$$\text{OCxR}/(\text{PRy} + 1) \times 100\%, \tag{9.1}$$

where PRy is the period register of the timer.
If the fault pin (OCFA for OC1-OC4 and OCFB for OC5) drops low, the read-only fault status bit OCxCONbits.OCFLT is set to 1, the OCx output is set to high impedance, and an interrupt is generated if the interrupt enable bit is set. The fault condition is cleared and PWM resumes once the fault pin goes high and the OCxCONbits.OCM bits are rewritten.
You can use the fault pin with an Emergency Stop button that is normally high but drops low when the user presses it. If the OCx output is driving an H-bridge that powers a motor, setting the OCx output to high impedance will signal the H-bridge to stop sending current to the motor. An emergency stop button will likely have other requirements (such as also physically cutting power to the motor), depending on your application.

0b110 PWM mode with fault pin disabled. Identical to above, except without the fault pin.

0b101 Dual compare mode, continuous output pulses. When the module is enabled, OCx is driven low. OCx is driven high on a match with OCxR and driven low on a match with OCxRS. The process repeats, creating an output pulse train. An interrupt can be generated when OCx is driven low.

0b100 Dual compare mode, single output pulse. Same as above, except the OCx pin will remain low after the match with OCxRS until the OC mode is changed or the module is disabled.

0b011 Single compare mode, continuous pulse train. When the module is enabled, OCx is driven low. The output is toggled on all future matches with OCxR, until the mode is changed or the module disabled. Each toggle can generate an interrupt.

[1] Initialize OCxR before enabling the OC module to handle the first PWM cycle. After enabling the OC module, OCxR is read-only.

0b010 Single compare mode, single high pulse. When the module is enabled, OCx is driven high. OCx will be driven low and an interrupt optionally generated on a match with OCxR. OCx remains low until the mode is changed or the module disabled.

0b001 Single compare mode, single low pulse. When the module is enabled, OCx is driven low. OCx will be driven high and an interrupt optionally generated on a match with OCxR. OCx will remain high until the mode is changed or the module disabled.

0b000 The output compare module is disabled but still drawing current, unless OCxCONbits.ON = 0.

OCxR, x = 1 to 5 If OCxCONbits.OC32 = 1, then all 32-bits of OCxR are compared against Timer23's 32-bit count. Otherwise, only OCxR⟨15:0⟩ is compared to the 16-bit count of Timer2 or Timer3, depending on OCxCONbits.OCTSEL.

OCxRS, x = 1 to 5 In dual compare mode, if OCxCONbits.OC32 = 1, then all 32-bits of OCxRS are compared against Timer23's 32-bit count. Otherwise, only OCxRS⟨15:0⟩ is compared to the 16-bit counter Timer2 or Timer3, depending on OCxCONbits.OCTSEL. In PWM mode, the value of this register is transferred into OCxR at the beginning of each period; therefore, modifying this register sets the next duty cycle. This SFR is unused in the single compare modes.

Timer2, Timer3, or Timer23 (depending on OCxCONbits.OC32 and OCxCONbits.OCTSEL) must be separately configured. Output compare modules do not affect the behavior of the timers; they simply compare the timer count to values in OCxR and OCxRS and alter the digital output OCx on match events.

The interrupt flag status and enable bits for OCx are IFS0bits.OCxIF and IEC0bits.OCxIE, and the priority and subpriority bits are IPCxbits.OCxIP and IPCxbits.OCxIS.

PWM modes

The Output Compare modes you are most likely to use are the PWM modes. They can be used to drive H-bridges powering motors or to continuously transmit analog values represented by the duty cycle. Microchip often equates "duty cycle" to the duration OCxR of the high portion of the PWM waveform, but it is more standard to refer to the duty cycle as a percentage, 0 to 100%. A plot of a PWM waveform is shown in Figure 9.2.

9.3 Sample Code

9.3.1 Generating a Pulse Train with PWM

Below is sample code using OC1 with Timer2 to generate a 10 kHz PWM signal, initially at 25% duty cycle and then changed to 50% duty cycle. The fault pin is not used.

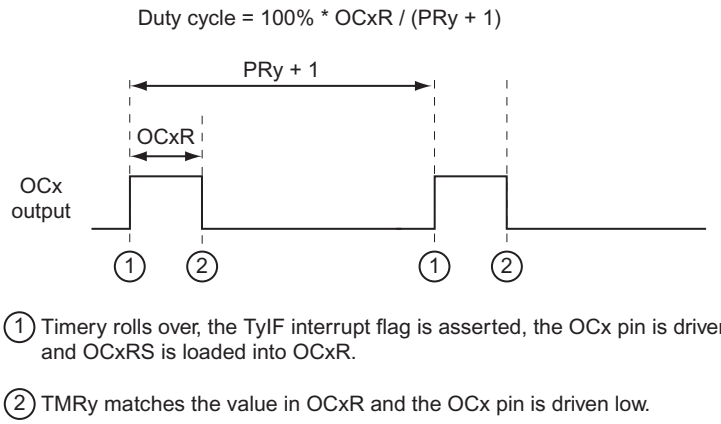

Figure 9.2
A PWM waveform from OCx using Timery as the time base.

Code Sample 9.1 `OC_PWM.c`. **Generating 10 kHz PWM with 50% Duty Cycle.**

```
#include "NU32.h"          // constants, functions for startup and UART

int main(void) {
  NU32_Startup();          // cache on, interrupts on, LED/button init, UART init

  T2CONbits.TCKPS = 2;     // Timer2 prescaler N=4 (1:4)
  PR2 = 1999;              // period = (PR2+1) * N * 12.5 ns = 100 us, 10 kHz
  TMR2 = 0;                // initial TMR2 count is 0
  OC1CONbits.OCM = 0b110;  // PWM mode without fault pin; other OC1CON bits are defaults
  OC1RS = 500;             // duty cycle = OC1RS/(PR2+1) = 25%
  OC1R = 500;              // initialize before turning OC1 on; afterward it is read-only
  T2CONbits.ON = 1;        // turn on Timer2
  OC1CONbits.ON = 1;       // turn on OC1

  _CP0_SET_COUNT(0);       // delay 4 seconds to see the 25% duty cycle on a 'scope
  while(_CP0_GET_COUNT() < 4 * 40000000) {
    ;
  }
  OC1RS = 1000;            // set duty cycle to 50%
  while(1) {
    ;                      // infinite loop
  }
  return 0;
}
```

9.3.2 Analog Output

DC analog output

Low-pass filtering a high-frequency, constant duty cycle PWM signal can create an approximately constant analog output. The low-pass filter essentially time-averages the high and low voltages of the waveform,

```
average voltage = duty cycle * 3.3 V
```

assuming that the output compare module swings between 0 and 3.3 V (the range may actually be a bit less).

There are many ways to build circuits to low-pass filter a signal, including *active* filter circuits using op amps. Here we focus on a simple *passive* RC filter, shown in Figure 9.3 and described in Appendix B.2. The voltage V_C across the capacitor C is the output of the filter. When R is zero, the output compare module attempts to source or sink enough current to allow the capacitor voltage to exactly track the nominal PWM square wave, and there is no "averaging" effect. As the resistance R is increased, however, the resistor increasingly limits the current I available to charge or discharge the capacitor, meaning that the capacitor's voltage changes more and more slowly, according to the relationship $dV_C/dt = I/C$.

The charging and discharging of the capacitor, and its relationship to the product RC, is shown in Figure 9.4. RC low-pass filters are discussed in more detail in Appendix B.2.

In Figure 9.4, the RC filter voltage variation during one PWM cycle is rather large. To reduce this variation, we would choose a larger product RC by increasing the resistance R and/or capacitance C. The drawback of a large RC is that the filter's average output voltage changes slowly in response to a change in the PWM duty cycle. While this is not an issue if the desired

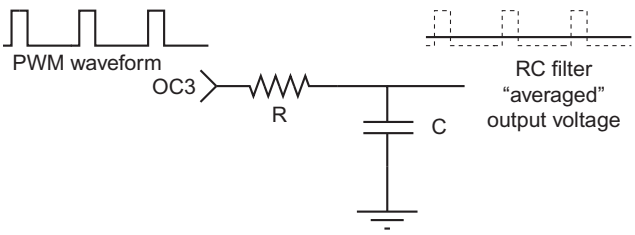

Figure 9.3
An RC low-pass filter "averaging" the PWM output from OC3.

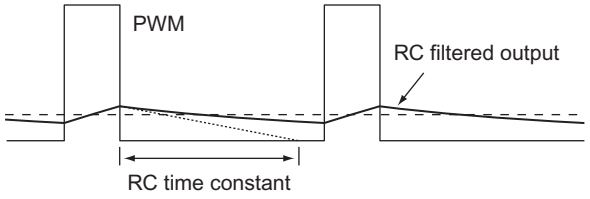

Figure 9.4
A close-up of the PWM, the RC filter output (with RC charging/discharging time constant illustrated), and the true time-averaged output (dashed). If the variation in the RC filtered output is unacceptably large, a larger value of RC should be chosen.

analog voltage is DC (constant), it is an issue if we want the analog voltage to vary in time, as discussed below.

The PWM OCxR value can range from 0 to PRy + 1, where PRy is the period register of the Timery base for the OCx module. This means that PRy + 2 different average voltage levels are achievable.

Code Sample 9.2 generates a PWM signal at 78.125 kHz with a duty cycle determined by OC3R in the range 0-1024. The timer base is Timer2. With an RC filter with a suitably large time constant attached to OC3, the voltage across the capacitor reflects the DC analog voltage requested by the user.

Code Sample 9.2 `OC_analog_out.c`. **Using Timer2, OC3, and an RC Low-pass Filter to Create Analog Output.**

```
#include "NU32.h"          // constants, functions for startup and UART

#define PERIOD 1024        // this is PR2 + 1
#define MAXVOLTAGE 3.3     // corresponds to max high voltage output of PIC32

int getUserPulseWidth(void) {
  char msg[100] = {};
  float f = 0.0;

  sprintf(msg, "Enter the desired voltage, from 0 to %3.1f (volts): ", MAXVOLTAGE);
  NU32_WriteUART3(msg);

  NU32_ReadUART3(msg,10);
  sscanf(msg, "%f", &f);

  if (f > MAXVOLTAGE) {   // clamp the input voltage to the appropriate range
    f = MAXVOLTAGE;
  } else if (f < 0.0) {
    f = 0.0;
  }

  sprintf(msg, "\r\nCreating %5.3f volts.\r\n", f);
  NU32_WriteUART3(msg);
  return PERIOD * (f / MAXVOLTAGE);  // convert volts to counts
}

int main(void) {
  NU32_Startup();         // cache on, interrupts on, LED/button init, UART init

  PR2 = PERIOD - 1;       // Timer2 is OC3's base, PR2 defines PWM frequency, 78.125 kHz
  TMR2 = 0;               // initialize value of Timer2
  T2CONbits.ON = 1;       // turn Timer2 on, all defaults are fine (1:1 divider, etc.)
  OC3CONbits.OCTSEL = 0;  // use Timer2 for OC3
  OC3CONbits.OCM = 0b110; // PWM mode with fault pin disabled
  OC3CONbits.ON = 1;      // Turn OC3 on
  while (1) {
    OC3RS = getUserPulseWidth();
  }
  return 0;
}
```

Time-varying analog output

Suppose we want to create a sinusoidal analog output voltage, such as

$$V_a(t) = 1.65\text{ V} + A\sin(2\pi f_a t),$$

by changing the duty cycle of the PWM. The frequency of this desired analog output is f_a. Now we have three relevant frequencies: the PWM frequency f_{PWM}, the RC filter cutoff frequency $f_c = 1/(2\pi RC)$ (Appendix B.2.2), and the desired analog voltage frequency f_a. Examining the frequency response of the low-pass RC filter in Figure B.9(a), we can adopt the following rules of thumb for choosing these three frequencies:

- $f_{PWM} \geq 100 f_c$: The PWM waveform consists of a DC component, a base frequency at f_{PWM}, and higher harmonics to create the square wave output. According to the gain response of the filter, only about 1% of the magnitude of the PWM frequency component at $100 f_c$ makes it through the RC filter.
- $f_c \geq 10 f_a$: Again consulting the RC filter frequency response, we see that signals at frequencies ten times less than f_c are relatively unaffected by the RC filter: the phase delay is only a few degrees and the gain is nearly 1.

For example, if the PWM is at 100 kHz, then we might choose an RC filter cutoff frequency of 1 kHz, and the highest frequency analog output we should expect to be able to create would be 100 Hz. In other words, we can vary the PWM duty cycle through a full sinusoid (e.g., from 50% duty cycle to 100% duty cycle to 0% duty cycle and back to 50% duty cycle) 100 times per second.[2] If the desired analog output is not sinusoidal, then it should be a sum of signals at frequencies less than 100 Hz.

The maximum possible PWM frequency is determined by the 80 MHz PBCLK and the number of bits of resolution we require for the analog output. For example, if we want 8 bits of resolution in the analog output levels, this means we need $2^8 = 256$ different PWM duty cycles. Therefore the maximum PWM frequency is 80 MHz/256 = 312.5 kHz.[3] On the other hand, if we require $2^{10} = 1024$ voltage levels, the maximum PWM frequency is 78.125 kHz. Thus there is a fundamental trade-off between the voltage resolution and the maximum PWM frequency (and therefore the maximum analog output frequency f_a). While higher resolution analog output is generally desirable, there are limits to the value of increasing resolution beyond a certain point, because the device receiving the analog input may have a limit to its analog input sensing resolution and the transmission lines for the analog signal may be subject to electromagnetic noise that creates voltage variations larger than the analog output resolution.

---

[2] Note that this creates a signal that is the sum of a 100 Hz sinusoid with a duty cycle amplitude equal to 50% plus a DC (zero frequency) component of amplitude equal to 50% duty cycle.

[3] Technically this yields 257 possible duty cycle levels, since OCxR = 0 corresponds to 0% duty cycle and OCxR = 256 corresponds to 100% duty cycle.

9.4 Chapter Summary

- Output compare modules pair with Timer2, Timer3, or the 32-bit Timer23 to generate a single timed pulse or a continuous pulse train with controllable duty cycle. Microcontrollers commonly control motors using pulse-width modulation (PWM) to drive H-bridge amplifiers that power the motors.
- Low-pass filtering of PWM signals, perhaps using an RC filter with a cutoff frequency $f_c = 1/(2\pi RC)$, allows the generation of analog outputs. There is a fundamental tradeoff between the resolution of the analog output and the maximum possible frequency component f_a of the generated analog signal. If the PWM frequency is f_{PWM}, then generally the frequencies should satisfy $f_{PWM} \gg f_c \gg f_a$.

9.5 Exercises

1. Enforce the constraints $f_{PWM} \geq 100 f_c$ and $f_c \geq 10 f_a$. Given that PBCLK is 80 MHz, provide a formula for the maximum f_a given that you require n bits of resolution in your DC analog voltage outputs. Provide a formula for RC in terms of n.
2. You will use PWM and an RC low-pass filter to create a time-varying analog output waveform that is the sum of a constant offset and two sinusoids of frequency f and kf, where k is an integer greater than 1. The PWM frequency will be 10 kHz and f satisfies 50 Hz $\geq f \geq$ 10 Hz. Use OC1 and Timer2 to create the PWM waveform, and set PR2 to 999 (so the PWM waveform is 0% duty cycle when OC1R = 0 and 100% duty cycle when OC1R = 1000). You can break this program into the following pieces:
   a. Write a function that forms a sampled approximation of a single period of the waveform

$$V_{out}(t) = C + A_1 \sin(2\pi ft) + A_2 \sin(2\pi kft + \phi),$$

   where the constant C is 1.65 V (half of the full range 0 to 3.3 V), A_1 is the amplitude of the sinusoid at frequency f, A_2 is the amplitude of the sinusoid at frequency kf, and ϕ is the phase offset of the higher frequency component. Typically values of A_1 and A_2 would be 1 V or less so the analog output is not saturated at 0 or 3.3 V. The function takes f, A_1, k, A_2, and ϕ as input and creates an array `dutyvec`, of appropriate length, where each entry is a value 0 to 1000 corresponding to the voltage range 0 to 3.3 V. Each entry of `dutyvec` corresponds to a time increment of 1/10 kHz = 0.1 ms, and `dutyvec` holds exactly one cycle of the analog waveform, meaning that it has $n = 10\text{ kHz}/f$ elements. A MATLAB implementation is given below. You can experiment plotting waveforms or just use the function for reference. A reasonable call of the function is `signal(20, 0.5, 2, 0.25, 45)`, where the phase 45 is in degrees. An example waveform is shown in Figure 9.5.

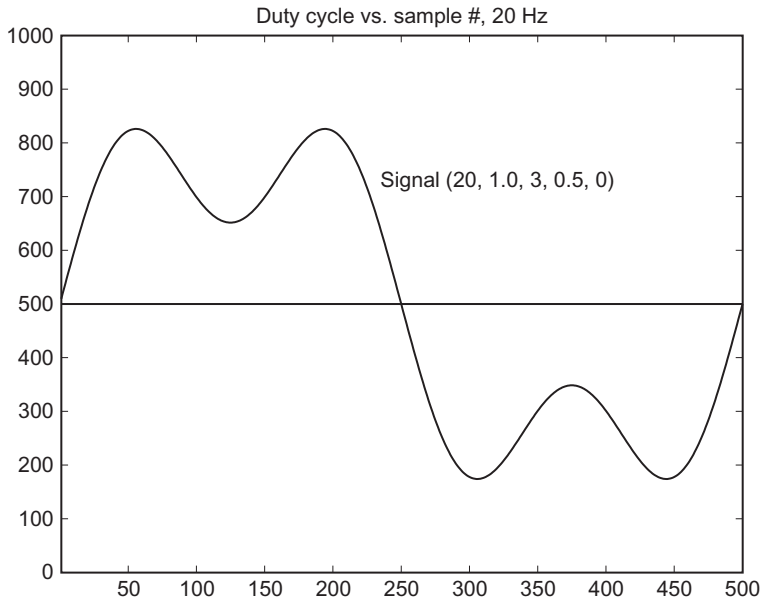

Figure 9.5
An example analog output waveform from Exercise 2, plotted as the duration 0 to 1000 of the high portion of the PWM waveform, which has a period of 1000.

```
function signal(BASEFREQ,BASEAMP,HARMONIC,HARMAMP,PHASE)

% This function calculates the sum of two sinusoids of different
% frequencies and populates an array with the values.  The function
% takes the arguments
%
% * BASEFREQ:  the frequency of the low frequency component (Hz)
% * BASEAMP:   the amplitude of the low frequency component (volts)
% * HARMONIC:  the other sinusoid is at HARMONIC*BASEFREQ Hz; must be
%              an integer value > 1
% * HARMAMP:   the amplitude of the other sinusoid (volts)
% * PHASE:     the phase of the second sinusoid relative to
%              base sinusoid (degrees)
%
% Example matlab call:  signal(20,1,2,0.5,45);

% some constants:

MAXSAMPS = 1000;     % no more than MAXSAMPS samples of the signal
ISRFREQ = 10000;     % frequency of the ISR setting the duty cycle; 10kHz

% Now calculate the number of samples in your representation of the
% signal; better be less than MAXSAMPS!

numsamps = ISRFREQ/BASEFREQ;
if (numsamps>MAXSAMPS)
```

```
  disp('Warning: too many samples needed; choose a higher base freq.');
  disp('Continuing anyway.');
end
numsamps = min(MAXSAMPS,numsamps);  % continue anyway

ct_to_samp = 2*pi/numsamps;         % convert counter to time
offset = 2*pi*(PHASE/360);          % convert phase offset to signal counts

for i=1:numsamps  % in C, we should go from 0 to NUMSAMPS-1
  ampvec(i) = BASEAMP*sin(i*ct_to_samp) + ...
              HARMAMP*sin(HARMONIC*i*ct_to_samp + offset);
  dutyvec(i) = 500 + 500*ampvec(i)/1.65; % duty cycle values,
                                         % 500 = 1.65 V is middle of 3.3V
                                         % output range
  if (dutyvec(i)>1000) dutyvec(i)=1000;
  end
  if (dutyvec(i)<0) dutyvec(i)=0;
  end
end

% ampvec is in volts; dutyvec values are in range 0...1000

plot(dutyvec);
hold on;
plot([1 1000],[500 500]);
axis([1 numsamps 0 1000]);
title(['Duty Cycle vs. sample #, ',int2str(BASEFREQ),' Hz']);
hold off;
```

b. Write a function using the NU32 library that prompts the user for A_1, A_2, k, f, and ϕ. The array `dutyvec` is then updated based on the input.
c. Use Timer2 and OC1 to create a PWM signal at 10 kHz. Enable the Timer2 interrupt, which generates an IRQ at every Timer2 rollover (10 kHz). The ISR for Timer2 should update the PWM duty cycle with the next entry in the `dutyvec` array. When the last element of the `dutyvec` array is reached, wrap around to the beginning of `dutyvec`. Use the shadow register set for the ISR.
d. Choose reasonable values for RC for your RC filter. Justify your choice.
e. The main function of your program should sit in an infinite loop, asking the user for new parameters. In the meantime, the old waveform continues to be "played" by the PWM. For the values given in Figure 9.5, use your oscilloscope to confirm that your analog waveform looks correct.

Further Reading

PIC32 family reference manual. Section 16: Output compare. (2011). Microchip Technology Inc.