# PropKeyer – Expanded Project Documentation

## Contents

1. System Overview and Design Objectives .................................. 2
 1.1 Modes of Operation ................................................ 2

2. Hardware Description ............................................................... 4
 2.1 Main Electronics Assembly ...................................... 4
 2.2 Top Panel Assembly ................................................ 4
 2.3 Side Panel Assembly ................................................ 4
 2.4 Rear Panel Connectors ............................................. 5
 2.5 Code Speed Control ................................................. 5
 2.6 DC Power Requirements ........................................... 6

3. Software Description ............................................................... 7
 3.1 Cogs and Methods in the PropKeyer ....................... 7
 3.1.1 Cogs: ......................................................... 7
 Main ...................................................... 7
 fpcontrols ............................................... 8
 sidetone ................................................. 8
 keyboard ................................................ 8
 padstate ................................................. 8
 dot ........................................................ 9
 dash ...................................................... 9
 3.1.2 Methods: .................................................... 9
 iambicA ................................................. 9
 iambicB ................................................. 9
 bug ....................................................... 10
 sideswi .................................................. 10
 console .................................................. 10
 ddtx ...................................................... 10
 pushstack ................................................ 10
 popstack ................................................. 10

# 1. System Overview and Design Objectives

The PropKeyer is an electronic keyer designed to generate Morse Code characters. It's primarily intended to be used with an "iambic" key (horizontal-acting dual-paddle key), but can emulate different styles of code sending that have come and gone as the telegrapher's art has developed over the past 150 years. This instrument allows the operator to re-create the feel of Morse code telegraphy as it used to be – while using a modern day iambic key. The PropKeyer also supports a PS/2 keyboard and a conventional "straight" single-lever manual telegraph key.

The platform for this project is the Propeller Development Board. The wealth of resources available there and on the Propeller chip itself makes it easy to implement additional convenience features, such as:

- A six-level FIFO stack running in its own cog stores any key presses that come ahead of the actual transmission and inserts them at the proper time for perfectly formed characters.

- Another cog runs a software-generated side tone oscillator which follows the keyer output. A small loudspeaker and a volume control with an on-off switch allows the user to hear the code as it is being sent out the rear-panel key jack. This cog also controls a LED on the front panel for a visual indication of keyer output.

- Non-volatile calibration data for the speed controller is stored in the unused portion of the EEPROM provided on the Development Board.

## 1.1 Modes of Operation

The modes of operation emulated by the PropKeyer are:

- Iambic A – A series of dots or dashes is transmitted when the paddle is pushed to either the right or left, respectively. (This is the usual setup for a right-handed person. Some left-handers find it convenient to reverse the function of the paddles.) If both paddles are pressed simultaneously, alternating dots and dashes are sent.

- Iambic B − This mode is nearly identical with iambic A, with one subtle difference: At the end of the alternating dot-dash series that occurs when both paddles are pressed, an extra code element is sent which is the opposite of the element that was in progress when the paddles were released. In other words, if a dot was in progress when both paddles are released, a dash is sent, and vice-versa.

- Bug – also referred to as a semi-automatic or "Vibroplex" key, after the company that originally manufactured and patented this design around 1904 (and,

incidentally, is still manufacturing them today). The bug key has, in effect, a single set of electrical contacts and features a horizontal arm, or pendulum, which transmits a series of dots when the paddle is pushed to the right and held. The speed is controlled by a sliding weight on the vibrating arm. When the arm is pushed to the left, the contacts are simply closed and the operator forms the individual dashes by finger action. An arrangement of springs keeps the arm centered when no code is being sent. True bug keys are strictly mechanical and, unlike the iambic keys, do not require an electronic keyer to function. The bug appeared at a fairly early stage in the development of telegraphy and was widely adopted when it became evident that a side-to-side motion resulted in faster sending and less operator fatigue.

- Sideswiper – The sideswiper (also known as a "sidewinder", or "cootie" key) was another early departure from the vertical acting straight key. This key is mechanically simpler and so less expensive than the Vibroplex bug. It has a single paddle which can be moved to either side and simply closes a set of contacts, regardless of which direction the paddle is moved. The length of both dots and dashes is under control of the operator who simply slaps the paddle back and forth.

- Console, or keyboard mode – The PropKeyer accepts a PS/2 computer keyboard. When this mode is selected characters are fully formed and transmitted by typing on the keyboard.

# 2. Hardware Description

## 2.1 Main Electronics Assembly

The hardware platform on which this unit is constructed is the Propeller Development Board, manufactured by Parallax, Inc. This board makes up the main electronics assembly of the PropKeyer. It's approximately 3 by 4 inches and is supplied with a Propeller chip, EEPROM, 5 MHz crystal, two voltage regulators and four pins for USB access and programming the chip. Most of the board consists of an unpopulated prototyping area.

The Propeller chip has a unique design consisting of a central hub processor with eight peripheral processors which may be assigned to independent tasks. Due to the wealth of resources available on this chip, no other integrated circuits are required to complete the PropKeyer hardware design.

The enclosure for this unit is a 4 x 7 x 2.5 inch cabinet with a sloping front. The Propeller Development Board is mounted inside this enclosure, with the power input connector and programming pins accessible through the rear of the cabinet.

## 2.2 Top Panel Assembly

A top panel assembly holds the operating controls: a five-position rotary switch for selecting the operating mode, a rotary potentiometer for setting the code speed and a pushbutton connected directly in parallel with the output jack – useful when tuning up the transmitter. There are also two LEDs – one which blinks to follow the Morse code being transmitted while the other indicates that power is present. This top assembly is mounted inside the sloping front panel of the cabinet and connects to the main board by means of a 20-conductor ribbon cable and a dual-row header.

## 2.3 Side Panel Assembly

The aforementioned loudspeaker and volume control are mounted on the side panel of the cabinet and connect to the main board via a two-pin header. The rotary volume control activates a switch at the extreme low end of its range to shut the 'speaker off completely.

## 2.4 Rear Panel Connectors

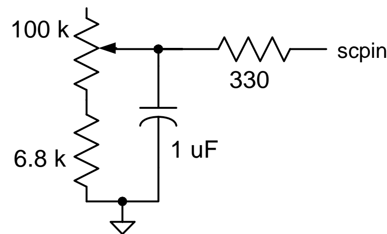
All I/O connectors are mounted on the rear panel assembly

| Function | Connector Type |
| --- | --- |
| Iambic paddle key | 3.5 mm stereo audio jack |
| Straight key | 3.5 mm mono audio jack |
| Keyer output jack | ¼ inch mono 'phone jack |
| PS/2 keyboard | 6-pin mini-DIN, female |

The keyer output jack is driven by an open-collector NPN transistor. This transistor is a general-purpose switching type which can sink over 100 milliamperes and switch an open-circuit voltage up to +40volts. This is compatible with most modern solid-state low-power transceivers using "positive" keying. This transistor is not suitable for transceivers which use "negative" keying or for older, tube-based gear. A keying relay is usually necessary to handle the negative voltage and/or higher voltage requirement imposed by this type of equipment.

## 2.5 Code Speed Control

The code speed is set with an analog potentiometer located on the top panel assembly and calibrated in words per minute. A numerical value is derived from the resistance of the pot by using the circuit shown below, connected to pin *scpin*.




A measure of the resistance of the 100 kohm potentiometer is obtained by driving *scpin* high for 1 millisecond to charge the capacitor, then reversing the direction of the pin and using a counter to record the time for the capacitor voltage to decay to the logic threshold. This decay time is proportional to the resistance of the external circuit.

Two momentary-contact pushbutton switches mounted on the Development Board allow the user to enter calibration data for the speed control pot. When the "SC5" button is pressed, the setting of the speed control pot at that time is stored in EEPROM. This value determines the lower code speed limit (5 words per minute). The "SC40" button does the same operation, storing the pot setting in EEPROM for the upper code speed limit (40 wpm). The "CW" LED on the top panel flashes when the pot setting has been read and successfully stored into the EEPROM.

The calibration procedure only requires the user to rotate the pot to its minimum limit before pressing SC5 and to its maximum limit before pressing SC40. These values are used within the *fpcontrols* cog to calculate the correct value of wpm for any intermediate position of the speed control pot. (See *fpcontrols* in the Software Description section of this document.)

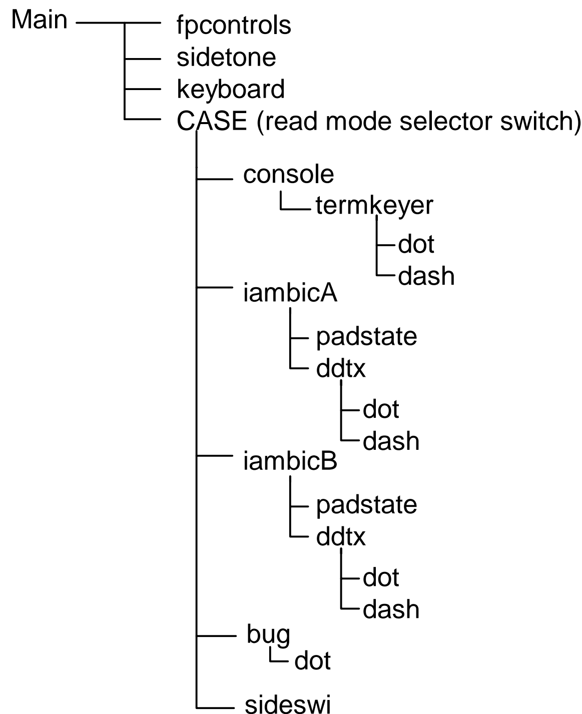
### 2.6 Dc Power Requirements

Power requirement for the PropKeyer is +8 to +16 volts dc at 60 mA and is supplied to the system via the dc coax connector mounted on the Propeller Development Board. The board was modified slightly to allow a SPST rocker switch, mounted on the side of the cabinet, to switch power to the board. A Schottky diode in series with this switch prevents accidental reversed power connection to the board .

# 3. Software Description

The PropKeyer software package is written entirely in the SPIN language and relies heavily on the multi-tasking capability afforded by the eight independent processors available on the Propeller chip. Some of the methods (i.e., sub-routines, or modules) used here are taken directly from the Propeller library and Object Exchange available on the Parallax web site. PS/2 keyboard support and the methods to access the EEPROM for non-volatile storage of calibration data are examples of these.

A schematic of the program structure is shown below:




## 3.1 Cogs and Methods in the PropKeyer Object

### 3.1.1 Cogs:

***Main***

- Initialize I/O pin directions.
- Initialize EEPROM interface method.
- Initialize speed control pot limits.
- Start front panel controls cog (*fpcontrols*).
- Start *sidetone* cog.
- Start serial data terminal interface cog.
- Start keyboard support cog.

- Initialize dash weight to 3 (default value).
- Enter loop monitoring the state of the mode selector switch on the front panel. The method corresponding to the selected mode is called and program control is returned here when the mode switch changes state.

***fpcontrols***

- Set up the counter register for monitoring the speed control pot setting.
- Initialize I/O pin directions.
- Start loop:
  - update the speed control pot setting, *decaycount*
  - if new user-defined speed control pot limit settings, store in EEPROM
  - copy speed limits from EEPROM into *scpot5* and *scpot40*
  - calculate updated value for variables *wpm* and *timeunit*

  A number corresponding to the speed control pot resistance is stored in variable *decaycount* and is converted to a number between 5 and 40 (corresponding to words per minute) by SPIN code calculation:

  *wpm* := 35*(decaycount - scpot5)/(scpot40 - scpot5) + 5

  Based on the "PARIS" standard method of measuring words per minute, the basic Morse code time unit in milliseconds is equal to 1224/wpm. This is the ON time duration of one dot. In system clock cycles it is given by:

  *timeunit* := (1224/wpm) * (clkfreq/1000)

***sidetone***

- Initialize I/O pin directions.
- Continuously monitor the state of the PropKeyer output pin (keypin). Blink the “CW” LED on the top panel assembly and generate a square wave at an audio frequency whenever the output *keypin* is high. The sidetone audio is directed to the sidetone output pin, *sidepin*. The program constant *tonefreq* sets the output frequency in Hertz.

***keyboard***

This library method runs in its own cog. It receives characters typed on a PS/2 computer keyboard and returns the corresponding ASCII code. This method is called from within the *console* method.

***padstate***

This cog monitors the state of the iambic paddle key contacts every 100 microseconds. De-bouncing is accomplished by left shifting the state of a paddle key into a four-bit register, *dotdebo* or *dashdebo*, depending on which paddle has been pressed. When the debounce register contains %1000, we assume that a high-to-low transition has occurred on the pin in question, indicating a paddle

press. Flags *dotpress* and/or *dashpress* are set TRUE. These are global flags and must be cleared by the object that uses them, such as *ddtx*.

***dot*** (Note: This cog is started from methods *ddtx* or *bug*)

- Set pin *keypin* as output
- Set global flag *dotflag* TRUE
- Set *keypin* high for one time unit
- Set *keypin* low for one time unit
- Set *dotflag* FALSE
- Stop cog

***dash*** (Note: This cog is started from method *ddtx*)

- Set pin *keypin* as output
- Set global flag *dashflag* TRUE
- Set *keypin* high for *dashweight* time units
- Set *keypin* low for *dashweight* time units
- Set *dashflag* FALSE
- Stop cog

### 3.1.2 Methods:

***iambicA***
***iambicB***
These methods continuously call a CASE statement which monitors the state of the iambic key paddles.

If only a single paddle is pressed,

- Clear the debounce register associated with the key pressed
- Start the *padstate* cog
- Set a flag, *ddsem*, indicating which paddle, dot or dash, is pressed
- Call method *ddtx*, which transmits a dot or dash according to setting of *ddsem*
- Pop the FIFO stack and send dots or dashes stored there to *ddtx*
- Stop the *padstate* cog

If both paddles pressed,

- Alternate flag *ddsem* on each pass through the loop and call *ddtx*.
- When paddles are released, send an additional element if in iambic B mode.
- Check the mode switch and if it has changed state return to the Main method

***bug***
Start loop:
- If the dot paddle is closed, send a dot by starting the *dot* cog.
- If the dash paddle is closed, send continuous high level out to *keypin*.
- Check the mode switch and if it has changed state return to the main method.

***sideswi***
- If either paddle is closed, send a continuous high level out to *keypin.*
- If both paddles are closed, or neither paddle is closed, send low level to *keypin.*
- Check the mode switch and if it has changed state return to the Main method.

***console***

construct the ASCII-to-Morse lookup table. This table is an array of word-length (two byte) variables, indexed by ASCII code. Each table entry holds the Morse symbol length (number of dots and dashes) in byte 1 and the dot-dash pattern for the symbol in byte 0. The pattern in byte 0 is left justified, dots being indicated by a “0” bit and dashes by a “1”.

Start loop:
- Get character input from PS/2 keyboard.
- Convert lower case to upper.
- If Ctrl-w followed by a number, update *dashweight* variable.
- If Ctrl-f followed by a number, update Farnsworth timing variable, *farnsext.*
- Check the mode switch and if it has changed state return to the Main method.

***ddtx***
- Send out a dot or dash, depending on the value of flag *ddsem*; a dot if flag is TRUE, a dash if FALSE.
- While the element is in progress, capture any further paddle contacts by monitoring the flags *dotpress* and *dashpress* (set by cog *padstate*).
- Push dot or dash request onto the FIFO stack.

***pushstack***

Find bottom of stack and insert new data there. Stack entries are 5, 10 or 9999 (dot, dash, End Of Stack mark, respectively).

***popstack***

Remove a data item from the top of the stack and move all other entries up one place.