

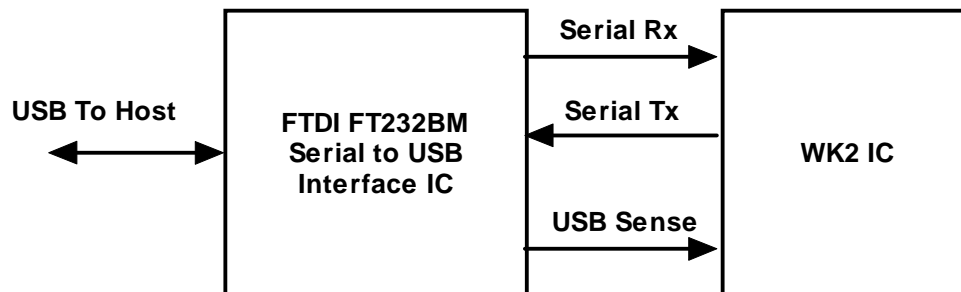
Introduction

WK_USB is a new version of the original WinKey CW keyer for Windows. In addition to hardware improvements, WK_USB incorporates a new microcontroller called WK2. One of the design goals for WK2 was to maintain backwards compatibility to WK1 so that existing WK1 applications could hook up to WK2 and operate with no modifications. WK_USB is the name of the new kit that incorporates the WK2 chip. This document will cover the WK_USB kit functionality.

- 1200 Baud Serial Rx/Tx Interface through USB VCP port (virtual serial port)
- Iambic CW Paddle Interface with separate dit and dah inputs
- Two separate output ports each with:
 - Key Output Port
 - PTT Output Port
- 128 character input buffer
- Four message pushbuttons
- Paddle only sidetone mode
- Optional battery power
- HSCW and QRSS Capability
- Dedicated Sidetone Output
- Standalone Enhanced K12 Keyer Mode

USB Interface

Since serial ports are all but obsolete on new PCs, WK_USB talks to the host PC over a USB interface. WKUSB uses an FTDI FT232BM interface IC which, through it's associated driver, makes WK_USB look like a standard serial com port. It is very simple to connect to WK_USB to an existing WK1 application. In fact, no modifications are required to existing WK1 applications except possibly allowing access to a com port number greater than COM4. This is because USB VCP com ports usually install as COM7 or COM8, but can be reassigned to a lower number.

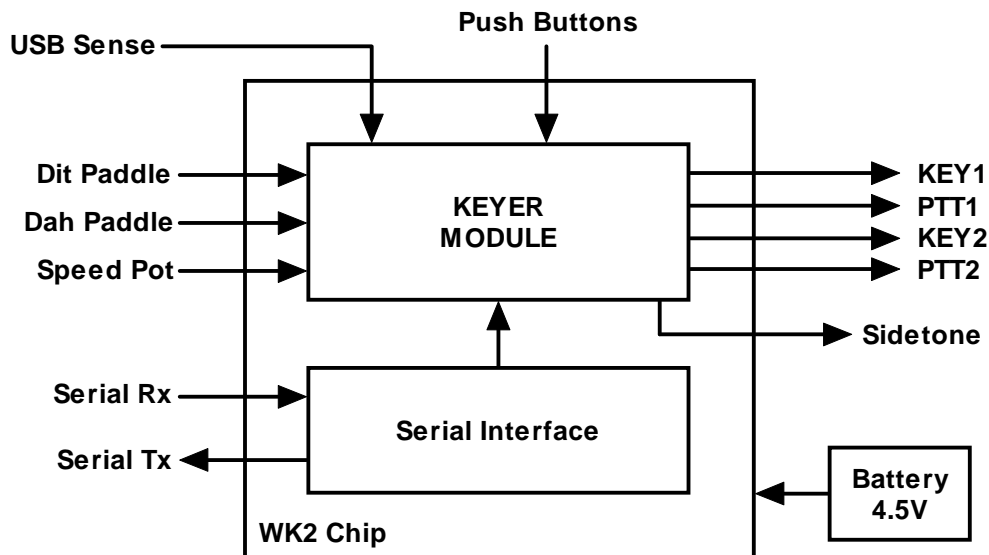


The block diagram above shows how the standard serial connection on the WK2 IC is fed to the FTDI chip which essentially converts it to USB format. The FTDI USB driver installed on the host PC presents the interface to Windows applications as a standard serial port so that the MSCOMM driver, or other serial driver, can be used.

Standalone Mode

A significant addition to WK2 is an integrated standalone keyer. This allows WK2 to be battery powered and used when not connected to a PC application. The following block diagram shows the components of the WK2 chip. Unlike WK1, the keyer module is not dependent on the serial interface for normal operation. The default state for the keyer is to

run in standalone mode whereby the user makes setting changes using the pushbuttons and entering commands on the paddles. When WK2 is connected to a host PC it remains in standalone mode until the host issues an open command.



WK1 to WK2 Software Compatibility

No modifications are required to use WK2 with your application. WK2 reports itself with a version number of 20 or higher so your application at least has to be able to accept this. All WK1 commands are supported on WK2 so no functionality will be lost. WK2 does offer some new features that are worth while taking advantage of and they will be covered in this document.

Adding WK2 Support to your Application

Here are the fast track steps to add WK2 support.

- 1) **Check Version:** WK2 will return the version number when the open command is issued just like WK1, if the return value is 20 (decimal) or higher then WK2 is connected.
- 2) **WK2 only accepts ADMIN commands when closed:** This should be a non-issue, WK1 would accept commands even when the interface was closed ! No application should operate that way so this should not impact your application.
- 3) **New Recommended Initialization Procedure:** The original open procedure outlined for WK1 will work for WK2 but this one will make transitioning back and forth to standalone operation much smoother. This procedure will still work fine for WK1.
 - 1) Open the serial communication interface (same as WK1)
 - 2) Delay 400 ms to allow WK1 to init from power up
 - 3) Send four 13H null commands, this syncs WK's command parser with the host
 - 4) Use the echo command to be sure that something is connected, if a WK doesn't respond with in 2 seconds abort and present error to user.
 - 5) Issue the open command and read back the WK version
 - 6) WK is now ready to use.
 - 7) Load WK's settings either by block or individually.

WKUSB doesn't require delays like WK1 did, if you plan to support both WK1 and WK2 you need a single 400 msec delay after asserting the RTS and DTR lines to allow WK1 to finish its power up init. (See init example the end of this document) A soft reset command is not needed, the only reason you would have to issue a soft reset would be to reset WK back into standalone mode with stored defaults. You can issue a soft reset and it will do no harm but it is unnecessary.

- 4) **PINCFG Register Changes:** The PINCFG register is more flexible in WK2. (see SetPinConfig command on page 8 of the WK2 manual) First of all, sidetone can be enabled or disabled independently of the KEY/PTT setting. You just set the Sidetone Enable bit to enable sidetone and clear it to disable sidetone.

In WK2, PTT is always enabled and there is no reason to turn it off since it is a separate output now. The PTT enable bit is not used and is ignored by WK2.

WK2 supports two output ports. The KeyOut 1 and KeyOut 2 enable bits are used to select the output port you want to use. You can select both at the same time although this really is not too useful.

The upper four bits in the PINCFG register control keyer mode and are unchanged.

Note that you can still use the PINCFG in the same manner as WK1, settings that worked for WK1 will work for WK2. The exception is that PTT will be on all the time.

- 5) **Pushbutton Interface:** You can tell WK_USB to notify your application when one of its pushbuttons has changed state. To do this you have to enable the pushbuttons with a new ADMIN command. (see SetWK2Mode on WK2 manual page 5, and Request WK2 Status on page 12) When pushbuttons are enabled, the return status byte is redefined. What was the TUNE bit in WK1 mode is the PB change flag. When this bit is set the status byte presents the current state of the pushbuttons. When a WK open command is issued, pushbutton reporting is always disabled. This insures that a WK1 application will not accidentally have PB reporting enabled. This means that if you want PB reporting you have to enable it every time you open WK_USB.
- 6) **Paddle Only Sidetone:** WK2 has a new feature called paddle only sidetone. It is enabled by setting the MSB of the sidetone control byte. (see WK2 manual page 6) When this bit is set sidetone is played for paddle entry and inhibited for serial data sent from the host.
- 7) **Serial Input Buffer Size:** The serial input buffer size is 128 bytes in WK2 vs. 32 bytes in WK1. You may want to take advantage of this to improve communications between your application and WK2. Buffer pointer commands will operate on the full 128 byte buffer.
- 8) **Be sure to Close WK_USB when your done!** When your application is closed be sure to issue a close command to WK_USB, this is required to place WK_USB back in standalone mode. The user may want to leave his keyer connected to the PC and use it independently of a PC application. This wasn't as important in WK1 since WK1 would lose power as soon as the com interface was closed. With WK_USB, since it can be battery powered, the host needs to tell WK_USB it is finished. If your existing app does not issue a close command it's not a big deal, there is a way to close the interface from the WK_USB by pressing and holding the command pushbutton until WK_USB resets to standalone mode.

Standalone Considerations

Standalone operation adds a level of complexity to the system that didn't exist in WK1. Although some folks have put batteries in WK1, it was never officially supported and had problems. Here's a list of things to know about standalone operation.

- 1) When WK_USB is powered up for the very first time it loads a default set of standalone parameters.
- 2) These parameters can be modified through standalone paddle commands or by hooking WK_USB up to the Windows application WK2MGR.
- 3) If the user modifies parameters in standalone mode and wants to preserve them, they need to remember to save the settings with the standalone **P** command.
- 4) When WK_USB is connected to the USB port, the PC recognizes the USB device and registers it but WK_USB remains in standalone mode until an application opens it. The only serial commands WK_USB will accept while in standalone are ADMIN commands.
- 5) WK2MGR works by reading and writing the EEPROM in the WK2 IC. The EEPROM contains all of the standalone settings and local messages. Since the EEPROM is accessed by ADMIN commands, WK_USB can remain in a closed state.
- 6) When an application opens WK_USB, WK_USB leaves standalone mode and will no longer responds to the command pushbutton making it impossible to enter paddle commands.
- 7) In order to proceed smoothly, the settings in WK_USB must match those in the controlling application. Otherwise, WK settings shown in the application's dialog boxes may not agree with what is in the WK2 chip. There are two ways to resolve this, first the host can overwrite all of WK2's settings with the host's settings, this is the way WK1 works. The second way is to read WK2's settings out of EEPROM and load the application's data structures with these values before continuing. This is probably not the logical way to do it but it can be done. The ADMIN command called READ EEPROM allows this. WK2 will return a 256 byte data block that is defined by the structure template shown below.
- 8) You can use the new ADMIN command WRITE EEPROM to transfer new settings to WK2 EEPROM, or like WK1, you can use the LOAD DEFAULTS command to load them. As a third option you can selectively write parameters with individual commands. K1EL recommends using the LOAD_DEFAULTS command.
- 9) When WK_USB is closed, standalone operation is restored and whatever is in EEPROM is reloaded into the active state. It's possible to load new settings into WK2's EEPROM before closing so that the settings don't change when the app is closed. It might be a good idea to offer this as an option to the user. In other words a pushbutton to copy the current state to WKUSB EEPROM. The flexibility is there, it is up to the application to decide what makes the most sense.
- 10) An easy way to handle all of this is to go along with the notion that there is a standalone state and a host state. When your app opens WK_USB it loads in new settings and it's ready to go. When you are all done you close WK_USB and the original standalone settings are restored from EEPROM.

Speed Pot Handling

The pot interface is tricky due to its dual mode nature. It can run in either pot lock or non-pot lock mode.

Pot Lock Mode

In pot lock mode the speed pot directly sets the sending speed, the host is passive in this mode and simply displays the current WPM setting to the user. The host obtains the current WPM by either polling for it using the Get Speed Pot command or through the WK status bytes that are sent to the host whenever the speed pot is moved.

Setup Speed Pot command vs. Pot Lock Mode.

The Setup Speed Pot command specifies the minimum WPM and the WPM range of the speed pot. It works this way, if you set the minimum to 5 and the range to 30 then the speed pot will swing from 5 WPM to 35 WPM, full counterclockwise to full clockwise. One thing to remember in Pot Lock mode is that if the minimum or range values are changed, the speed pot value will be adjusted accordingly. For example, let's say the speed pot is set at midscale, with WPM min set to 5 WPM and WPM range set to 20 WPM. The speed pot value will be halfway between 5 and 25 which is 15 WPM. If you redefine the limits to be 10 WPM min and a 40 WPM range, the speed pot will now return a value of 30 WPM. $(40/2 + 10)$. It's important to remember that if you change the min or range values you **MUST** reset the speed pot to zero to force WK to update the speed pot to be within the new limits. Follow this procedure when issuing the Set Speed Pot command:

- 1) Issue the Set Up Speed Pot command, setting min WPM and WPM range
- 2) Issue Set WPM to zero (to force an immediate speed pot update)
- 3) Issue Read Speed Pot command to obtain the newly updated speed pot value

This keeps everything in sync nicely. If you decide to leave pot lock mode and issue an arbitrary WPM, there are more things to consider...

Non-Pot Lock Mode

In this mode the host can arbitrarily set the WPM rate to whatever it wants. If the speed pot is turned, the new speed pot value is returned to the host but no change is made to WK's speed directly. It's up to the host to decide what to do with the speed pot in this mode. The min WPM and WPM range values still govern the value of the returned speed pot value and also the value that the host is allowed to set. If the host tries to set a value outside of the set WPM limits, it will be ignored. The thing to remember in non-pot lock mode is that you have to keep the WPM range in mind when you set the speed.

Automatic WPM Clamping

In non-pot lock mode if the host uses the Setup Speed Pot command to set a speed range less or greater than the WPM value currently set, WK will automatically move the speed within range. As an illustration, let's say the min WPM was set to 10 WPM, the range set to 30, and the current speed was set to 35. If a Setup Speed Pot command was issued that changed the WPM range to 20, WK would move the current WPM down to 30 to keep it within the range window. That's why it's always important to read the speed pot back after adjusting the WPM window and present the value to the user. In addition, in non-pot lock mode you have to then send the value back to WK to complete the process.

So, the rule for non-pot lock mode is to always call get pot after setting the min or range values and then send that value back to WK with a set speed command. Your app has to be well behaved and not change the speed to something outside the min and range values or you will get into trouble.

Don't Mix Pot Lock and Non-Pot Lock Modes

One way WK apps get in trouble is to change modes in midstream without considering the range settings. It's best to choose a mode to run the speed pot in and leave it in that mode.

EEPROM Format

```
/** Winkey EEPROM Block **/  
  
typedef struct {  
    BYTE magic;        // 0x00  
    BYTE modereg;      // 0x01  
    BYTE oprrate;      // 0x02  
    BYTE stconst;      // 0x03  
    BYTE weight;       // 0x04  
    BYTE lead_time;    // 0x05  
    BYTE tail_time;    // 0x06  
    BYTE minwpm;       // 0x07  
    BYTE wpmrange;     // 0x08  
    BYTE xtnd;         // 0x09  
    BYTE kcomp;        // 0x0a  
    BYTE farnswpm;     // 0x0b  
    BYTE sampadj;      // 0x0c  
    BYTE ratio;        // 0x0d  
    BYTE pincfg;       // 0x0e  
    BYTE k12mode;      // 0x0f  
    BYTE cmdwpm;       // 0x10  
    BYTE freeptr;      // 0x11  
    BYTE msgptr1;      // 0x12  
    BYTE msgptr2;      // 0x13  
    BYTE msgptr3;      // 0x14  
    BYTE msgptr4;      // 0x15  
    BYTE msgptr5;      // 0x16  
    BYTE msgptr6;      // 0x17  
    BYTE msgs[MAXMSG]; // 0x18  
}WKEEPROM, *LPWKEEPROM;
```

Magic value is a constant 0xA5.

Other parameters match up with values as described in the WK2 manual.

There are two additional bytes that are used by standalone operation only:

1) k12mode is formatted as follows:

- Bit 2 Enable msg brk by paddle (standalone only)
- Bit 3 Select 0 and 9 cut for serial number (standalone only)

2) cmdwpm sets the WPM rate used for paddle entry commands in standalone mode.

The remaining data comprises the standalone message store. There are six message slots which are not fixed lengths. The msgptrN bytes point to the location of a message in the storage array. freeptr points to the first location of free message memory. If a message pointer is set to zero the message slot is empty. Otherwise the location of the message is that value offset from the msgs array start. When all memory is used freeptr will be set to zero.

I discourage modifying the message store from a host application. The message store is small and there are only six (four directly accessed in WKUSB) messages. Better to store messages on the PC where you have more space and flexibility.

WK Init Psuedo Code, error handling not shown for clarity

```
// define data structure
HANDLE hComm;
DCB dcb;

// First open the serial port
hComm = CreateFile (portStr, GENERIC_WRITE|GENERIC_READ, 0, NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);

// Get the current comm. State

GetCommState(hComm, &dcb);

// Set new state 1200 baud, one stop bit, no parity, 8 bits, enable DTR, disable RTS

dcb.BaudRate = CBR_1200;
dcb.StopBits = ONESTOPBIT;
dcb.Parity = NOPARITY;
dcb.ByteSize = 8;
dcb.fDtrControl = DTR_CONTROL_ENABLE;
dcb.fDsrSensitivity = FALSE;
dcb.fOutX = FALSE;
dcb.fInX = FALSE;
dcb.fNull = FALSE;
dcb.fRtsControl = RTS_CONTROL_DISABLE;

// Set new comm. state
SetCommState(hComm, &dcb);

// Load timeout values into structure

// Specify time-out between characters for receiving.
comTimeOut.ReadIntervalTimeout = 0;

// Specify value that is multiplied
// by the requested number of bytes to be read.
comTimeOut.ReadTotalTimeoutMultiplier = 0;

// Specify value is added to the product of the
// ReadTotalTimeoutMultiplier member
comTimeOut.ReadTotalTimeoutConstant = 250; // init w/250ms timeout

// Specify value that is multiplied
// by the requested number of bytes to be sent.
comTimeOut.WriteTotalTimeoutMultiplier = 0;

// Specify value is added to the product of the
// WriteTotalTimeoutMultiplier member
comTimeOut.WriteTotalTimeoutConstant = 0;

// Set the new timeout values.
SetCommTimeouts(hComm,&comTimeOut);
```

```
        Sleep (400);                // delay in case we have WK1 attached

// Purge receive port

        PurgeComm(hComm, PURGE_RXCLEAR)

//declare some variables

        BYTE buf[8];
        BYTE version;

// Send three null commands to resync host to WK2

        wk_send (NULLIMM_CMD);
        wk_send (NULLIMM_CMD);
        wk_send (NULLIMM_CMD);

// Echo Test to see if WK is really there
        wk_send (ADMIN_CMD);
        wk_send (ADMIN_ECHO);
        wk_send (0x55);

// Read back echo, will timeout if nothing attached.

        if (wk_read() == 0x55) {

                // Open WK
                wk_send (ADMIN_CMD);
                wk_send (ADMIN_OPEN);

                // WK will return its version number

                version = wk_read();

                // Now set a more reasonable timeout
                comTimeOut.ReadTotalTimeoutConstant = 10; // 10 ms timeout
                SetCommTimeouts(hComm,&comTimeOut);
        }

// Winkey init complete
```