Winkey Interface Guide

Steve Elliott K1EL

February 16, 2005

This document will describe the recommended procedure for initializing Winkey from a host Windows application and some general guidelines for writing an effective driver for Winkey.

A WIN32 DLL is available for Winkey interfacing and it is recommended to use this dll if possible since it will make your interfacing task much easier.

Initialization

Winkey V9 and later require a slightly different init than older rev parts. This is due to the fact that it takes WK v9 slightly longer to initialize. The following procedure will work for all parts:

1) Open serial communications port. Use 1200 baud, 8 data bits, no parity
2) To power up WK enable DTR and disable RTS
3) Delay for ½ second to allow WK to finish init
4) Purge the receive port to make sure there are no stray Rx characters sitting in the buffer. This is done by simply reading the port until empty.
5) Issue a calibration command:
   Byte 0: ADMIN_CMD
   Byte 1: ADMIN_CAL
   Byte 2: 0xFF
6) Check to make sure WK is attached and operational
   Byte 0: ADMIN_CMD
   Byte 1: ADMIN_ECHO
   Byte 2: 0x55
   Now read Rx until 0x55 is returned or you reach a 1 second timeout, if you timed out that means WK is not connected, the serial port selection is incorrect, there is something wrong with the cable or WK itself is no working.
7) Now it's time to officially open the WK interface, this is done by the following command sequence:
   Byte 0: ADMIN_CMD
   Byte 1: ADMIN_OPEN
   WK will respond with its firmware revision byte to let you know it opened correctly. If it does not respond with in ½ second (very unlikely if the echo command completed properly) something went wrong and needs to be addressed before continuing.

8) Now that Winkey is open, the returned status handler must be enabled. WK will return unsolicited status bytes as a part of its normal operation. To write an effective interface these bytes must be received and processed. There are three types of returned status:
    a) Change in WK status byte
    b) New speed pot position
    c) Echoed letter sourced either from a serial transmission or paddle entry

    The most effective way to handle returned status is with a thread that is activated when a character has been received from WK. The thread will process the returned byte and update global variables that the main program can access. For example, your application needs to know when WK has asserted XOFF so that it doesn't send any characters and overflow WK's input FIFO (this is a particularly bad thing to do)

    Placing the receive processing in a parallel thread allows the main program to simply check a bit in the global status register to see if WK is busy. Otherwise it would have to read back WK's status byte every time it wanted to send a byte to WK. Threading can also insure orderly transmits as well, more on this later.

9) Now it's time to load the operating state. The recommend way to do this is with the load defaults command which is a block load:
    Byte 0: DFLTS_CMD;
    Byte 1: modereg
    Byte 2: speed
    Byte 3: stconst
    Byte 4: weight
    Byte 5: leadin
    Byte 6: tail
    Byte 7: minwpm
    Byte 8: wpmrange
    Byte 9: xtnd
    Byte 10: kcomp
    Byte 12: farns
    Byte 13: sampadj
    Byte 14: ditdahratio
    Byte 15: pincfg
    Byte 16: potrange
    Byte 17: GETPOT_CMD
    Byte 18: GETSTAT_CMD
    Notice that two commands are tacked on the end of the block load, this will kick off WK's initial status reporting and insure that the host's global copy of WK's status and speed pot position are up to date.

10) The init is now completed and WK can now accept and process commands. When the host application closes, it is a good idea to close the serial interface and enable RTS and disable DTR to power WK off. The WK close command is optional since if power is turned off WK will disappear anyway.

General Interface Guidelines

The top three things that a host interface should never do is:

1) Overflow the input FIFO. This can happen if the host continues to send bytes to WK after WK has asserted the XOFF bit in the status register. So please make sure you monitor this bit with the least possible latency, that's why a thread based Rx routine is so important.
2) That said there is one time when it is ok to send after XOFF is asserted, that is if XOFF asserted in the midst of sending a buffered command. For example let's say you sent a buffered speed command and WK asserted XOFF, the host needs to ignore XOFF and send the parameter to the speed command. There will always be enough slop to accept several bytes after XOFF asserts. The rule is to never let WK sit waiting for a buffered command parameter.
3) Do not send commands or data from two places in your application. Due to the multi threaded operation of windows it is highly possible that two processes can attempt to send to WK at the same time and commands can get interleaved with data and WK can do unexpected things. Although the command parser in WK is robust, there are certain byte sequences that can cause it to lock up. An example is if two transmit streams collide and WK misinterprets a zero parameter as an ADMIN command, that means that whatever byte follows next can send WK into reset, calibration, closed, or diagnostic mode.

On the subject of multiple transmit streams, the WK DLL uses one thread to receive from WK and a second thread to transmit to WK. Any time the host app wants to send data to WK it is funneled to one transmit handler in the thread. A local FIFO is resident in the Tx thread process which can be as large as desired, the thread keeps track of the XON/XOFF status and also makes sure transmitted commands are atomic. Please have a look at the source to the WKDLL to see how this thread operates. There are other ways to do the same thing without threads using windows timer callbacks but they will not work as efficiently.

Email k1el@aol.com for the source to the WKDLL