A Guide to the Pertec Tape Controller Software

January, 2023

Chuck Guzis, Sydex Inc.

Introduction

This document serves as a guide to implementing microcontroller (“MCU”) firmware for the Pertec Tape Controller board (“PTCB”). This should not be construed as a strict set of rules, but merely guidelines, should a different microcontroller or firmware be desired.

As written, the firmware supplied was targeted at a specific MCU board, the STM32F4xx development board employing an ST Micro STM32F407 microprocessor. It was selected primarily because it was inexpensive, had more than adequate speed and memory and 5 volt tolerant GPIO, support for SD cards and full-speed USB OTG. It also has provisions for an Ethernet MAC and other features not used. A battery-backed real-time clock/calendar was a plus for time-stamping files.

Since the PTCB employs TTL-level logic, an MCU with 5 volt tolerant I/O is important, unless the implementer desires to add external level-shifting logic. It should be noted that the MCU is not directly connected to any of the tape interface pins, so drive/current sinking requirements are minimal.

As designed, the software allows for use of either asynchronous serial (UART) or USB (CDC ACM) interfaces, the choice being determined at compilation time.

Requirements

An essential understanding of the interface to the PTCB logic is important when selecting an MCU. Note should be taken of the following:

There are 4 8-bit GPIO interfaces on board:

One 8 bit bidirectional interface for data.

One 8 bit output interface for drive control.

One 8 bit input interface for status.

One 8 bit output interface for logic control.

Data interfaces employ latches to hold data going to and from the drive. The drive formatter itself has no handshake provisions; it signals when a byte is available for reading and when it has accepted a byte for writing. Flip-flops record the state of both types of traffic and can be interrogated in the status register. Some signals are transient during a data transfer, such as hard error status; they too, are made available in the status register.

Since an 8 bit interface to all registers is used, the logic control lines signal which latch is being addressed by the MCU. For example, there are 16 status signals, divided into two groups of 8 bits; the first are signals that should be interrogated during a data transfer; the second are static signals that can be checked after a transfer has been concluded.

If GPIO is at a premium, it should be possible to make use of only three 8-bit GPIO interfaces, with the drive control sharing the data interface. This has not be tried, but there appears to be no impediment toward using the interface in this way.

The logic control interface determines which 8 bit status register is selected for reading by the MCU, which 8 bit drive control register is selected, and the state of the data registers (empty or full). Additionally, a signal is provided to force the drive control registers into a high-impedance state.

For a call-out of the various tape interface signals, see the interface description in the PTCB hardware description document.

Status Registers

The status registers are configured thus; note that the status as read is negative logic (0 level = true):

Status register 0:

xxxx xxx1 = Read data parity bit; not used by most drives, which enforce odd parity.

xxxx xx1x = Data busy; corresponds to the IDBY interface signal.

xxxx x1xx = High speed mode; corresponds to the IHSP interface signal, used on streaming drives.

xxxx 1xxx = Read data available; signals that data has been latched; cleared by asserting TACK.

xxx1 xxxx = Write data buffer empty; signals that the drive has accepted data; cleared by TACK.

xx1x xxxx = File mark detected; cleared by asserting the GO control signal.

x1xx xxxx = Hard (uncorrectable) error detected; cleared by GO assertion.

1xxx xxxx = Corrected error detected; cleared by GO assertion.

Status register 1:

xxxx xxx1 = IRNZ; used by some drives to signal 800 bpi mode.

xxxx xx1x = End of tape detected; corresponds to IEOT on the tape interface.

xxxx x1xx = Drive is online; corresponds to IONL on the tape interface.

xxxx 1xxx = File protect; corresponds to IFPT on the tape interface.

xxx1 xxxx = Tape is rewinding; corresponds to IRWD on the tape interface.

xx1x xxxx = Tape is at load point (beginning); corresponds to ILDP on the tape interface.

x1xx xxxx = Drive is loaded and ready; corresponds to IRDY on the tape interface.

1xxx xxxx = Formatter busy; corresponds to IFBY on the tape interface.

Command Registers

The two command registers are negative-logic and configured thus:

Command register 0:

xxxx xxx1 = “GO” – when asserted, initiates data activity on trailing edge Also resets error status.

xxxx xx1x = Last word. Used during writes, must be asserted before the last byte is transferred

xxxx x1xx = Load; on some drives, pulsing this line initiates a tape load sequence

xxxx 1xxx = Reverse – when asserted with some operations causes reverse tape movement xxx1 xxxx = Rewind; when asserted, causes tape to rewind to load point

xx1x xxxx = Write; signifies a write operation

x1xx xxxx = Read threshold 1 – IRTH1 on the Pertec bus, use varies by manufacturer

1xxx xxxx = Read threshold 2 – IRTH2 on the Pertec bus, use varies by manufacturer

Command register 1:

xxxx xxx1 = Write file mark; when asserted with Write, causes a tape mark to be written.

xxxx xx1x = Erase, when asserted with Write, causes a long block gap to be written.
Can be used with other bits for non-write operations.

xxxx x1xx = Edit mode – used with other commands for various operations.

xxxx 1xxx = Rewind and unload – unloads tape and brings drive offline.

xxx1 xxxx = Initiate high-speed operation. May not be implemented on some drives.

xx1x xxxx = Drive address bit 1.

x1xx xxxx = Drive address bit 0.

1xxx xxxx = Formatter address; some drives combine this with the drive address to form a 3 bit drive select address.

It's important to note that the meaning of various command bit combinations can vary between drive models and manufacturers. The OEM manual for the drive should be considered to be the final authority on this.

Finally, there is an 8 bit logic control register configured as follows (so labeled on the PCB):

ENA = Command enable. A low level on this line enables the command register output. A high level puts the command register output into high-impedance mode.

DDIR = Data direction. A high level means that the MCU is in input mode and is ready to accept data from the read data register. A low level means that the MCU is prepared to send data to the write data register, strobed by a transition on LBUF (see below). Note that it is essential that the MCU data register direction be set accordingly.

LBUF = Load buffer. A low-to-high transition loads the write data register from the MCU data output.

SSEL = Status select. Selects between status register 0 and status register 1.

CSEL0 = Command select 0. A transition on this line loads command register 0 from the MCU.

CSEL1 = Command select 1. A transition on this line loads command register 1 from the MCU.

TACK = Transfer acknowledge. Pulsing this line resets both the 'Read Data Available' and 'Write Data Empty' status lines.

Operation Examples

Here follow some general illustrations of common functions.

1. Rewind tape – First check that the drive is online and not at load point (that is, not already rewound). Pulse the "Rewind" bit in the command register. When the bit is reset (high), the tape

drive begins rewinding and sets “rewinding” in the status register. Rewind is complete when the “rewinding” status is inactive and the drive reports online and load point.

2. Unload tape – Check if the drive is online, then pulse the “Unload” bit in the command register. When complete, the drive will become offline.

3. Read block – Check that the drive is online, Make sure that the MCU data register is set for input mode. Pulse the “Go” bit in the command register. Test and wait for “Formatter busy”, then wait for “Data busy”. When a data byte becomes available, the “Read Data Available” status register becomes active. Input and store the data byte in memory. Acknowledge the transfer by pulsing TACK in the logic control register. Continue the operation until “Data busy” goes inactive. Check the status register for “File Mark Detected” (the data count will be 0) as well as for “Hard Error” and “Corrected Error” status.

4. Write block – Check that the drive is online and that the “Protected” bit is inactive in the status register. Make sure that the MCU data register is set for output and set the DDIR logic control bit for output. “Prime” the data output register with the first byte of data pulsing “LBUF” and then assert “Go” and “Write” in the command register. Upon releasing “Go”, “Formatter busy” will go active; then “Data busy” will become active. When the drive has accepted the pre-loaded data byte, the “Write data buffer empty” will go active. The “TACK” flag should be reset and the next byte to be written should be loaded (“LBUF”) and the operation should be continued until the next-to-last data byte has been accepted by the drive. Assert “Last word” in the command register, then load the last data byte into the data output register (LBUF). After the drive has accepted the final data byte, “Data busy” will become inactive and the operation status can be examined.

5. Write file name – No data is transferred, but otherwise the operation is similar to “Write block” described above. The difference is that the “Write file mark” bit in the command register is set along with “Write” and “Go”. The operation will be complete when “Data busy” goes inactive in the status register.

A Practical Implementation

A sample MCU program for the STM32F407 accompanies this design. It is constructed using the gcc-arm-none-eabi C compiler and the libopencm3 ARM-cortex development suite. Firmware programming was accomplished by an ST-Link USB programming tool and the openocd programming software. Debian Linux was used as the host platform, but most 32- and 64-bit Linux or Windows installations should be adaptable. The project software is compiled and linked using a “make” script in a standard “Makefile”. Tape files are created and read in the SIMH .tap file format described at http://simh.trailing-edge.com/docs/simh_magtape.pdf.

What follows is a hierarchical description of the various sets of routines in the sample.

First level Execution begins in the **main.c** routine. It sets up the various devices and interfaces, namely:

1. Peripheral clocks and GPIO. Initial states of the various GPIO interface pins is established with regards to input vs. output, initial states, etc. This occurs in the **miscsubs.c** module.
2. If USB serial interface is being used the **usbcdc.c** routines are called to initialize connection with the host; if UART serial interface is compiled, the **uart.c** routines are called to establish the bit rate (normally 115,200).

3. The system “tick” clock is set up in **miscsubs.c** to provide an interrupt approximately every millisecond. After every 1024 milliseconds, an LED is toggled alternately on an off to provide a visual “I’m alive” indication. This millisecond counter is also used as a deadman timer in serveral routines, such as the ymodem interface.
4. A free-running timer counter is set up with a period of 500 nanoseconds. This is used to provide precise delays in establishing pulse widths for example. This timer is initialized in **miscsubs.c** also.
5. The operator input, wether UART or USB is polled for a “g” character to start operation.
6. The real-time clock is initialized, if necessary to reflect the current time of day and date and is implemented in **rtcsubs.c**. The MCU development board used in this example has a battery-backed RTC, so it should need to be initialized only when the time of day or date needs to be changed or when a new backup battery is installed. The RTC is used to date files and directories that are created.
7. The SDIO interface in **sdiosubs.c** is initialized and an attempt is made to mount any SD card that may be present. If none is inserted, it may be mounted manually later.
8. The tape drive interface is initialized to a known state by calling a routine in **tapedriver.c**. The drive itself may or may not be powered on or online at this point.
9. Control is passed to the command-line processor in **cli.c** and does not return.

The command-line processor is very simple—it accepts user input, tokenizes the input and calls the routine corresponding to the first token on the input line. The relevant routine is passed the list of the remaining tokens and, after processing the request, returns to the command processor loop. Part of the command prompt is a quick summary of tape drive status by calling a routine in **tapeutil.c**.

Second Level routines are invoked by the command processor. Some are very simple, such as the command help routine, which either displays a list commands or a brief description of a single command (given as an argument). Others deal with file system operations (located in **filesub.c**), tape operations (located in **tapeutil.c**), Ymodem file transmit/receive (located in **ymodem.c**) or real-time clock setting (located in **rtcsubs.c**).

Third Level routines are not directly invoked by the command processor, but service various requests from the second-level code. So, for example, while the code in **tapeutil.c** processes user requests for tape activities, it performs no direct operations on the tape interface. Rather, it calls various routines in t**apedriver.c**, which contains all of the lowest-level tape code. These third-level routines are as follows:

1. **comm.c** contains higher-level serial I/O code to perform formatted input or output, such as Uprintf which is a simplified printf clone. Depending on whether USB or UART communication is desired, the appropriate lower-level handler is called (either **usbcdc.c** or **uart.c**).
2. **ff.c** is the FATfs code developed by ChaN. It implements the basic filesystem for FAT, FAT32 and exFAT and can be obtained from elm-chan.org. It requires only a simple sector-level interface found in **diskio.c** for operation. Long filename support is provided by the **ffunicode.c** module. **Diskio.c** in turn, invokes the code in **sdiosubs.c**.
3. **sdiosubs.c** contains the driver code for accessing the microSD card contained on the MCU board. It employs 4-line SDIO and DMA to provide I/O activities that can overlap tape operations.
4. **filesub.c** contains mount, directory and basic I/O services; it interfaces to **ff.c**.
5. **tapedriver.c** implements low-level tape drive functions, among which are read or write block, writing file mark, rewind, unload, skip forward or reverse files or blocks, and obtaining drive status.

UART operation can be selected by compiling the MCU code with the symbol **USE_UART** defined.

A debugging facility may optionally incorporated by defining the symbol **SERIAL_DEBUG**. It provides simple I/O primitives (e.g. DBgets(), DBprintf()…) through serial UART1. It should not be used also with the **USE_UART** symbol defined, as both use the same device.

Thoughts for later implementation

Although the ymodem interface works well for file transfer, it does not operate at the full speed of the USB 2.0 interface. It might make more sense to add a composite USB driver facility implementing a mass-storage device (USB MSD) along with the USB CDC ACM device. Of course, this facility would not be implemented if the UART serial interface was selected.

An EBCDIC display option might be added to the **dump** command.